Highly Efficient Distributed Hypergraph Analysis: Real-time Partitioning and Quantized Learning

Wenkai Jiang
orcid.org/0000-0001-6791-4307

Submitted in total fulfilment of the requirements of the degree of

Master of Philosophy

School of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE

September 2018

Abstract

Hypergraphs have been shown to be highly effective when modeling a wide range of applications where high-order relationships are of interest, such as social network analysis and object classification via hypergraph embedding. Applying deep learning techniques on large scale hypergraphs is challenging due to the size and complex structure of hypergraphs. This thesis addresses two problems of hypergraph analysis, real-time partitioning and quantized neural networks training, in a distributed computing environment.

When processing a large scale hypergraph in real-time and in a distributed fashion, the quality of hypergraph partitioning has a significant influence on communication overhead and workload balance among the machines participating in the distributed processing. The main challenge of real-time hypergraph partitioning is that hypergraphs are represented as a dynamic hypergraph stream formed by a sequence of hyperedge insertions and deletions, where the structure of a hypergraph is constantly changing. The existing methods that require all information of a hypergraph are inapplicable in this case as only a sub-graph is available to the algorithm at a time. We solve this problem by proposing a streaming refinement partitioning (SRP) algorithm that partitions a real-time hypergraph flow in two phases. With extensive experiments on a scalable hypergraph framework named HyperX, we show that SRP can yield partitions that are of the same quality as that achieved by offline partitioning algorithms in terms of communication overhead and workload balance.

For machine learning tasks over hypergraphs, studies have shown that using deep neural networks (DNNs) can improve the learning outcomes. This is because the learning objectives in hypergraph analysis are becoming more complex these days, where features are difficult to define and are highly-correlated. DNNs can be used as a powerful classifier to construct features automatically. However, DNNs require high computational power and network bandwidth as the size of

DNN models are getting larger. Moreover, the widely adopted training algorithm, stochastic gradient descent (SGD), suffers in two main problems: vast communication overhead that comes from the broadcasts of parameters during the partial gradient aggregations, and the inherent variance between partial gradients, making the training process even longer as it impedes the convergence rate of SGD. We investigate these two problems in depth. Without sacrificing the performance, we develop a quantization technique to reduce the communication overhead and a new training paradigm, named cooperated low-precision training (C-LPT), in which importance sampling is used to reduce variance, and the master and workers collaborate together to make compensation for the precision loss due to the quantization.

Incorporating deep learning techniques into distributed hypergraph analysis shows a great potential in query processing and knowledge mining on high-dimensional data records where relationships among them are highly correlated. On one hand, such a process takes the advantage of strong representational power of DNNs as an appearance-based classifier; on the other hand, such a process exploits hypergraph representations to gain benefits from its strong capability in capturing high-order relationships.

Declaration

This is to certify that

1. the thesis comprises only my original work towards the MPhil except where indicated,
2. due acknowledgment has been made in the text to all other material used,
3. the thesis is less than 50,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Wenkai Jiang, September 2018

Acknowledgements

First and foremost, I would like to express the depth of my gratitude to my supervisors Professor Rui Zhang and Doctor Jianzhong Qi. They are brilliant researchers and advisers that lead me into this exciting field of developing techniques for distributed processing. They have always been very supportive during my study. Without their continuous help, trust, and encouragement, I would not have opportunities to pursue my own research interests and this degree would never be possible.

Next, I want to thank my advisory committee chair, Doctor Sean Maynard, for spending his time in giving me advice and guidance on building my study plan and providing valuable feedback on many matters.

I am also very grateful to my external advisor Doctor Wei Wang. I feel very fortunate to have him as an advisor as well as a friend. He not only taught me many technical skills in approaching research problems, but also helped me a lot in my life during the time I spent in Singapore. He gave me numerous constructive suggestions on quantized deep neural network training. I enjoyed those discussions with him and his insightful feedbacks have been invaluable to my research.

Thanks also due to all colleagues in our research group, especially Jin, Yiqing, Andy, Yu, Zeyi, Xiaojie, Chuandong, Saad, Jiazhen, Yuan, Gitansh, Han, He, Weihao, Xinting. They have been good friends in my life in Melbourne and many of them have provided me invaluable advice.

I own the best truly sincere gratitude to my parents, who have given me unconditional love and care all the time. Whenever I run into problems, they have always been trying to take me under their wings to make me warm and to help me get through. Without their support, I would not be able to achieve where I am today.

Last but not least, I am very thankful to the University of Melbourne, to the School of Computing and Information System, and to the Melbourne Research Scholarships, for providing the financial support for both of my education and the research in this thesis.

Preface

The work on scalable hypergraph learning, presented in Chapter 3, has been accepted for publication: Jiang, W., Qi, J., Yu, J., Huang, J., and Zhang, R. (2018). HyperX: A scalable hypergraph framework. IEEE Transactions on Knowledge and Data Engineering.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Hypergraphs have attracted many attentions during the past few years and have been applied to a wide range of applications such as social network analysis [132], image retrieval [55, 138], object classification [40], facial emotion recognition [56], and image segmentation [54]. Among these applications, data records are obtained from either online streaming or offline collections and modeled as hypergraphs for learning. Due to the huge success of deep learning techniques in many machine learning tasks, recent studies start combining deep neural networks (DNNs) with learning tasks on hypergraphs to achieve better performance. For example, deep learning has been used for creating embeddings of a social network represented as a hypergraph [87, 134]. This is to represent each vertex of a hypergraph in a latent lower dimensional space. After embeddings are created, DNNs can be applied once again to learn the relationships between these vertices from its own embeddings and embeddings of other vertices.

Thanks to the rapid development of the Internet, a huge amount of data is generated and collected at an unprecedented speed from all aspects of life, such as online social media, public health care systems or retail stores. The amount of available data in these areas has exploded significantly in the past decades also because of the fast growing number of applications and users. As a result, when modeling these data into hypergraphs, the size of hypergraphs has become larger. Analysis over these large scale hypergraphs poses new challenges in computational capabilities. It is common to use a cluster of distributed computers to solve learning problems on larger hypergraphs.

In this thesis, we investigate hypergraph processing and deep learning tasks on hypergraphs in the context of distributed computing. In particular, this thesis addresses two problems of hypergraph analysis, real-time partitioning and quantized neural networks training, in a distributed computing environment. Hypergraphs have been shown to be highly effective when modeling a

wide range of applications where high-order relationships are of interest. Applying deep learning techniques on large scale hypergraphs is challenging due to the size and complex structure of hypergraphs. For machine learning tasks over hypergraphs, studies have shown that using DNN can improve the learning outcomes. This is because the learning objectives in hypergraph analysis are becoming more complex these days, where features are difficult to define and are highly-correlated. DNNs can be used as a powerful classifier to construct features automatically. Hypergraph analysis using the combination of hypergraphs and DNNs can be found in many applications these days and achieves a remarkable success. For example, when detecting emotions of a person [56], facial images firstly pass through a convolutional neural network to be decomposed into several hidden expression features; next, high-order relationships between emotional features are depicted by hyperedges for emotion prediction. On one hand, such a process takes the advantage of strong representational power of DNNs as an appearance-based classifier; on the other hand, such a process exploits hypergraph representations to gain benefits from its strong capability in capturing high-order relationships. Incorporating deep learning techniques into distributed hypergraph analysis shows a great potential in query processing and knowledge mining on high-dimensional data records where relationships among them are highly correlated.

Hypergraphs have been used to represent the data records with full of rich structures and high-dimensional relationships among many applications. In these applications, the data records are represented by vertices and the relationships between data records are modeled as hyperedges. When applications involve huge amount of data, the size of hypergraph can be very large. Minimizing the query cost on such hypergraphs is crucial for the applications. For example, when querying a social network represented as a hypergraph for users' activities, real-time analytics requires low latency between sending queries and receiving results so that users do not experience a long waiting time. For a hypergraph representing millions of users and relationships, partitioning strategy is critical to reduce the latency. In this scenario, hypergraph partitioning helps to partition the query loads to several workers, which enables horizontal scaling of the large-scale hypergraphs.

Hypergraphs analysis requires first establishing a learning goal. However, as the structure of hypergraph is becoming more complicate, the size of hypergraph is getting bigger, and the application scenarios are becoming more diverse, it is becoming more difficult to establish these learning

Table 1.1: Hypergraph Applications

Application	**Algorithm**	**Vertex**	**Hyperedge**
Recommendation	[110]	Songs and users	Listening histories
Text retrieval	[52]	Documents	Semantic similarities
Image retrieval	[84]	Images	Descriptor similarities
Multimedia	[109]	Videos	Hyperlinks
Bioinformatics	[59]	Proteins	Interactions
Social mining	[111]	Users	Communities
Machine Learning	[120]	Records	Labels

goals. In particular, it is difficult to manually select or craft features on the hypergraphs as they are mostly highly related to each other. Deep neural networks (DNNs) can automatically generate these features by learning on a large number of hypergraphs. DNNs is a deep learning technique that has brought substantial advances to a wide range of applications that are driven by large-scale data sets and sophisticated models. However, training distributed DNNs is difficult to scale due to the inequality between computing time and communication time. The computation time can be significantly reduced by adding more workers to the cluster. However, the overhead of gradient synchronization increases dramatically along with the growth of number of workers [81]. The larger the scale of the distributed system, the more severe the bottleneck of the communication will be. Eventually this would offset the savings of computing power [80]. To tackle this communication bottleneck, model compression techniques [45, 57, 83, 91, 121, 136] such as sparse and quantized DNNs have been studied for *inference* tasks. The speed and efficiency of *inference* gain huge benefits from the use of modern hardware accelerators such as Google's TPU [63]. However, these accelerators are mainly used in *inference* but not *training* as the influence of reducing precision during *training* has not been fully investigated.

We investigate how to use the widely-deployed distributed cluster to realize real-time hypergraph partitioning, and achieve high scalability in DNNs training for hypergraph analysis. We propose techniques in hypergraph partitioning and neural network training to both ease the implementation and boost the computation. These techniques can be easily adopted by other distributed applications.

In the remainder of this chapter, we first describe the motivation of using hypergraph rather than normal graph to represent high-order relationships. Next, we elaborate the motivation of adopting quantization technique in deep neural networks training. After that, we briefly describe

the context of distributed computation over a cluster of commodity machines and its challenges. Last, the contributions of this thesis are summarized and the thesis outline is shown.

1.1 Motivation on Hypergraph Representations [1]

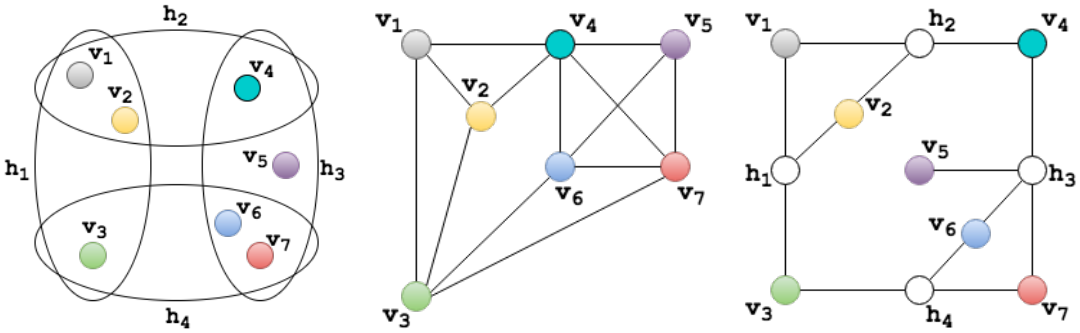
Graphs allow each edge to connect two vertices representing a certain relationship between them. For example, in a map of a city, an edges connecting two vertices may represent a path between two locations. In a wide range of applications, a relationship may be formed by more than two objects. For example, a picture posted by a user on a social network is likely to be liked by multiple of his friends; a tweet may be reposted by many users who have read it. In such applications, modeling objects and their relationships with a graph may incur information loss [134]. A common approach to address this problem is representing the objects and their relationships by vertices and *hyperedges* in a *hypergraph*. A hypergraph is a generalized graph where an edge can connect more than two vertices. Hypergraph models have shown great effectiveness in capturing high-order relationships [52,59,84,109–111,120]. Table 1.1 summarizes some representative examples of hypergraph applications.

1.1.1 Hypergraph Representation

We denote a hypergraph as $\mathcal{G} = \langle \mathcal{V}, \mathcal{H} \rangle$, where $\mathcal{V} = \{v_1, v_2, \ldots, v_m\}$ is a set of m vertices and $\mathcal{H} = \{h_1, h_2, \ldots, h_n\}$ is a set of n hyperedges. The *degree* of a vertex v, denoted by d_v, is the number of hyperedges that are incident to v. The *arity* of a hyperedge h, denoted by a_h, is the number of vertices in h, i.e., the number of vertices that are incident to h. Every vertex v and every hyperedge h is associated with some attributes of interest called a vertex value (e.g., a label), denoted by $v.val$ and a hyperedge value (e.g., a weight), denoted by $h.val$, respectively.

Both undirected and directed hyperedges are considered. An undirected hyperedge h is a nonempty subset of $\mathcal{V}$. For example, in Fig. 1.1a, there are four undirected edges, h_1, h_2, h_3 and h_4, represented by four ellipses. Each is a subset of $\mathcal{V} = \{v_1, v_2, \ldots, v_7\}$, e.g., $h_1 = \{v_1, v_2, v_3\}$. Since there are three vertices in h_1, the *arity* of h_1 is 3, i.e., $a_{h_1} = 3$. Meanwhile, since v_1 is in

[1] Part of this section has been published in: Jiang, W., Qi, J., Yu, J., Huang, J., and Zhang, R. (2018). HyperX: A scalable hypergraph framework. IEEE Transactions on Knowledge and Data Engineering.



(a) Hypergraph (b) CE (c) SE

Figure 1.1: Converting a hypergraph to a graph: CE and SE

both h_1 and h_2, its *degree* is 2, i.e., $d_{v_1} = 2$. A directed hyperedge h is a mapping on two disjoint nonempty vertex sets of $\mathcal{V}$: a source set $\mathcal{S}$ and a destination set $\mathcal{D}$, i.e., $h : \mathcal{S} \rightarrow \mathcal{D}$. For example, in Fig. 1.1a, we can change hyperedge h_1 to a directed hyperedge by assigning $\{v_1, v_2\}$ as the source set and $\{v_3\}$ as the destination set, i.e., $h_1 : \{v_1, v_2\} \rightarrow \{v_3\}$.

1.1.2 Conversion between Graphs and Hypergraphs

While applications of hypergraphs are emerging, there has been little work on developing a framework to support hypergraph representation directly. Graph frameworks cannot process hypergraphs without converting hypergraphs into graphs. Converting a hypergraph into a graph may inflate the size of the original hypergraph, because every hyperedge needs to be replaced by a clique which increases the number of edges and vertices. For example, a hypergraph studied previously [125] with 2 million vertices and 15 million hyperedges is converted to a bipartite with 17 million vertices and 1 billion edges. Such inflation causes huge difficulty in processing the hypergraph.

Two traditional graph representations are used for converting a hypergraph into a graph [134]: 1) *clique-expansion* (CE), which replaces each hyperedge with multiple edges forming a clique among the incident vertices of the hyperedges, and 2) *star-expansion* (SE), which replaces each hyperedge with a new vertex connected to its incident vertices. Fig. 1.1 illustrates these two approaches where the hypergraph in Fig. 1.1a is converted to a graph shown in Fig. 1.1b by CE and a graph shown in Fig. 1.1c by SE, respectively. Although these approaches are simple to implement, they have substantial limitations.

1. CE is inapplicable to algorithms that update hyperedge values as it no longer has records corresponding to the original hyperedges in the converted graph.

2. The converted graph may have orders of magnitude more vertices and edges compared with the original hypergraph. Fig. 1.1 shows a substantial growth even in a tiny hypergraph. The hypergraph with 4 hyperedges and 7 vertices is converted by CE into a graph with 13 edges and 7 vertices and by SE into a graph with 13 edges and 11 vertices.

3. For SE, there are two types of vertices, those from the original hypergraph and those converted from the hyperedges of the original hypergraph. Two vertex programs are used for updating these two types of vertices. When executing these two vertex programs, it takes two iterations to update the vertex values and hyperedge values, which is a drawback, because the two iterations double the overhead of updating the vertex replicas.

To partition a streaming hypergraph directly using hypergraph representation, we partition it using a recently proposed hypergraph framework, HyperX . HyperX is a thin layer built upon Apache Spark [128]. It provides flexible and expressive interfaces for the ease of implementation of hypergraph learning algorithms, operating directly on the hypergraph representation. To ease the use of the framework, HyperX provides a *hyperedge program* and a *vertex program* which are consistent with the *edge program* and *vertex program* used in popular graph frameworks such as GraphX. HyperX uses the Bulk Synchronous Parallel (BSP) message passing scheme, which is commonly used in synchronous graph processing frameworks.

HyperX builds a foundation that supports processing hypergraphs at large scale. When hypergraphs are large, HyperX distributes the computation over across many workers. This calls for a hypergraph partition algorithm to create partitions that can be processed in a distributed manner with a balanced workload and low communication costs among the workers. The efficiency of a hypergraph processing algorithm running on HyperX may be significantly impacted by the hypergraph partitions.

1.2 Motivation of Quantization in Deep Neural Networks

Hypergraphs have been shown to be highly effective when modeling a wide range of applications where high-order relationships are of interest. Applying deep learning techniques on large scale hypergraphs is challenging due to the size and complex structure of hypergraphs. For machine learning tasks over hypergraphs, studies have shown that using deep neural network (DNN) can improve the learning outcomes. This is because the learning objectives in hypergraph analysis are becoming more complex these days, where features are difficult to define and are highly-correlated. DNNs can be used as a powerful classifier to construct features automatically. Hypergraph analysis using the combination of hypergraphs and DNNs can be found in many applications these days and achieves a remarkable success. For example, when detecting emotions of a person [56], facial images are firstly passed through a convolutional neural network to be decomposed into several hidden expression features; next, high-order relationships between emotional features are depicted by hyperedges for emotion prediction. Another example is to create hypergraph embeddings using DNNs [87, 134]. This is to represent each vertex of a hypergraph in a latent lower dimensional space. After embeddings are created, DNNs can be applied once again to learn the relationships between these vertices from its own embeddings and embeddings of other vertices.

On one hand, such a process takes the advantage of strong representational power of DNNs as an appearance-based classifier; on the other hand, such a process exploits hypergraph representations to gain benefits from its strong capability in capturing high-order relationships. Incorporating deep learning techniques into distributed hypergraph analysis shows a great potential in query processing and knowledge mining on high-dimensional data records where relationships among them are highly correlated.

For distributed hypergraph analysis with deep learning techniques, the performance of the whole work flow depends not only on the hypergraph processing itself, but also on the performance of the DNNs, including phases of training and inference. Training distributed DNNs is known to be difficult to scale due to the inequality between computing time and communication time. Poor scalability will greatly damaged the efficiency of the whole system.

During the past few years, DNNs have been rapidly developed in various applications. Scaling up neural networks with respect to the number of parameters has significantly raised the state-of-the-art performance in several fields, including image classification, speech recognition, and arti-

ficial intelligence, such as AlphaGo [103] playing against professional players. The performance gain of these DNNs generally comes with high computational costs and large memory consumption, which may not be affordable for mobile platforms. Quantization is originally proposed to compress the deep neural network models in order to reduce the computation and storage costs of DNNs so that complex DNNs can be deployed on portable devices, such as mobile phones. Except for the need in portable device deployment, quantization can also be used to compress gradients to reduce communication overhead. In the training phase, the parallelization of stochastic gradient descent (SGD) requires synchronizations to gather gradients and parameters for aggregation in each iteration, and this introduces significant communication overhead. Using gradient quantization may reduce the communication overhead by tens of times.

If energy consumption is monitored for each operation when training DNNs, it is reported that communication counts for a significant fraction among various sources. Communication takes up to 50% of the power consumption in a multi-GPUs configuration of a state-of-the-art DNN training scheme [21]. This number includes the energy needed for data I/O on external and internal memories. It also includes energy for communicating values across the distributed systems.

The need to improve performance and reduce the communication overhead for DNNs is a hot research topic in recent years. The most popular approach is to use reduced precision representation for the numerical data computation. The most aggressive reduction in precision turns the whole model into a binarized neural network (BNN). BNN constrains both the weights and the activation to be either $+1$ or -1. It is claimed that these two values are extremely suitable for hardware optimization. Two different binarization functions have been proposed [24] to transform the full-precision variables into these two values.

Even though BNN utilizes binary weights and activation functions to compute gradients, the gradients that are aggregated to update the weights are in full precision. This is because it was believed that full-precision gradients are required for SGD to work properly. SGD explores the direction of gradients in small and noisy steps and the noise is offseted by the stochastic gradient aggregated in each step. As a result, it is critical to keep gradients in full precision. This actually is found not correct by recent work [82], which shows that gradients have very similar characteristic as weights, where *sign* matters more than *magnitude*. Thus, the gradients can be safely quantized as well.

Moreover, the noise of gradients provides a form of regularization that can help deep neural network models to generalize better. Quantization of the gradients is equivalent to adding more noise to the system. In this sense, previous techniques that have been widely adopted in modern DNNs training can be merged in without modifications, such as Dropout [106] and DropConnect [118].

An issue to be noticed is that, since the derivative of the sign function is zero almost everywhere, it can not be used for back-propagation (BP). Bengio et al. [10] study the problem of training stochastic discrete neurons using quantized gradients. The finding is that fastest training can be obtained by using the "straight-through estimator". All these factors motivate us to incorporate data quantization into the DNNs training and inference phase to make the operations more efficient on either desktop or portable devices.

1.3 Distributed Data Processing

Distributed computing on a cluster of machines can be deployed in different ways, e.g., the Message Parssing Interface (MPI), the Parallel Random Access Machine (PRAM), and MapReduce based platform (Hadoop [2]). In recent years, the MapReduce based platform becomes popular for applications with large scale of data that cannot be held by a single machine.

There are many implementations of the map-reduce computing paradigm. Hadoop is the most popular one among them, and it is more than just MapReduce. Hadoop provides three basic components: the storage engine called the Hadoop Distributed File System (HDFS); the resource manager named the Yet Another Resource Negotiator (YARN), and the computational paradigm, MapReduce. Apache Spark [3] can re-use the HDFS and YARN while it substitutes the MapReduce with a more advanced computational engine, which provides more operations including *filter, join*, and more importantly, it runs in memory, unlike Hadoop which is on disk. The three components in Hadoop is loosely-coupled, which makes them easy to be replaced by other counterparts correspondingly, such as Amazon S3 file system, Apache Mesos resource manager, Apache Hama computational framework. Plugging and unplugging different storage engines, resource managers, and computational engine is convenient. As an open source implementation, Hadoop provides an

[2] Apache Hadoop, Apache Software Foundation, available at https://hadoop.apache.org/
[3] Apache Spark, Apache Software Foundation, available at https://spark.apache.org/

assembly of distributed systems to make it look like an operating system running on a single machine. In the following subsections, we describe three computational paradigms that are widely used in distributed computing.

1.3.1 Computational Paradigms

Three basic components in Hadoop execute in a master-slave fashion to organize the cluster. This is to parallel the computation to all the slave machines at the first, and synchronize the result on master via network communication among the slaves. There are three main computation paradigms.

BSP Paradigm [116]. BSP contains a series of super-steps. In each super-step, only a subset of the data is used to compute. This subset of data firstly compute locally on every slaves, and then the result is combined with the messages received from last super-step to generate a bunch of new messages. Finally, these new generated messages are send to particular machines. After that, this cycle repeats. There is a synchronization step between each super-step, which guarantees that all messages are received from senders. Because of the synchronization, the overall speed depends on the slowest machine.

MapReduce [29]. Unlike BSP, MapReduce do not have super-step. On the other hand, each step of it has two phases, *map* and *reduce*. The *map* groups data records based on a user defined *key*, and the *reduce* aggregate data records sharing the same key based on a user defined reduce function. Both *map* and *reduce* executes on each machine independently. And there is one synchronization step between *map* and *reduce*.

Spark [128]. Different from above two paradigms, Spark use a directed acyclic graph (DAG) to determine the order of operations that will be executed on the partitioned data. And it adopts lazy evaluation to make real computation as late as possible. Each vertices in the DAG represents the operations that could be run independently on the distributed system and each edge connects two vertices to denote the source and destination of message passing over the network. There is one synchronization barrier between two connected vertices. Spark is faster than previous two by several magnitudes due to its memory running environment and lazy evaluation.

1.3.2 Challenges in Distributed Computing

When developing hypergraph processing algorithms and deep neural network training over a distributed system, the challenges lie in intensive computation, difficulty in balancing the workload, and communication overhead minimization.

Intensive Computation. A hypergraph with billions of vertices and hyperedges need to be partitioned to the distributed machines. Partitioning algorithms that have theoretical quality guarantee, e.g., semi-definite programming based solution, or good approximation properties, e.g., spectral clustering based solution, are prohibitive when confronting a large scale hypergraph [53]. For streaming hypergraph partitioning, only partial structure information is known. This makes partitioning even harder. So effective streaming hypergraph partitioning algorithm need to be designed.

As for deep neural networks (DNNs) training, a model can easily have millions parameters. Thanks to the remarkable development with GPU chips, the short of computational power has been remedied to some extents, but still as the model becomes larger, there is a great need in developing a low computational cost model.

Difficulty in Balancing Workload. Both hypergraph partitioning over Hadoop and DNNs training have synchronization barrier and need to wait for the slowest machine to complete the computation. Therefore, the balanced workload is essential to the efficiency of both applications. In hypergraph processing, because two hyperedges may overlap over several vertices, these vertices are therefore replicated several times. As a result, workload balancing in hypergraph partitioning not only need to consider the existing vertices in the hypergraph, but also need to optimize which vertices should be replicated and where to partition those replications to.

Communication Overhead. There is a trade-off between workload balancing and optimum communication. Take two extreme cases into consideration: there is no communication overhead if there is only one partition of data records, where the workload is totally unbalanced; there will be arbitrarily high communication overhead if data records are partitioned randomly into all workers, where the workload is absolutely balanced. When a hypergraph is partitioned, the fundamental rule is that partitioning should maintain the tightly nested sub-hypergraph so that natural clusters are partitioned near to each other. This would significantly reduce the unnecessary replicas and

communication overhead. Conversely, when training DNNs, this partitioning strategy does not work because there is no natural connection between data points. So to minimize communication overhead in DNNs training, we have to consider model compression or gradient quantization.

1.4 Thesis Contributions

We describe the contributions of this thesis in the real-time hypergraph partitioning on HyperX and the quantized deep neural network training. Regarding to the challenges discussed in Section 1.3.2, the communication challenges are intended to solve in both studies, while the computational and workload balance challenges are mainly reflected in the algorithm design in real-time hypergraph partitioning.

Our contributions on real-time hypergraph partitioning are:

- We investigate the real-time hypergraph partitioning problem where vertices arrive one at a time in a sequential manner. We formulate it as an integer programming problem to minimize the number of replicas during the partitioning, therefore minimize the communication costs when running hypergraph applications.

- We design a streaming refinement partitioning (SRP) algorithm which partitions a streaming hypergraph streaming in two steps. In the first step, rough partitioning, we investigate practical heuristics and propose a greedy strategy to create fast and rough partitions. In the second step, iterative refinement, we use label propagation with a fixed size sliding window to make the streaming partitioning algorithm independent of the streaming length in order to comply with the time and memory constraints in real-time processing.

- We evaluate SRP against a number of online and offline partitioning algorithms with extensive experiments on both real datasets and synthetic datasets. The results demonstrate that SRP is suitable for streaming partitioning as the average partitioning time is smaller than streaming rate. The results show that SRP not only deliver better partitioning results in terms of cut size and work-load balance compared to that of offline partitioning algorithms, but also delivered more efficient and effective performance when running hypergraph learning algorithms

Our contributions on quantized deep neural network training are:

- We investigate methods to incorporate deep learning techniques into distributed hypergraph analysis. We design a cooperated low precision training (C-LPT) paradigm for deep neural network training. In C-LPT, we allow masters and workers to keep two different sets of a model in different precision level. In each training iteration, the workers are trained on a low-precision model using a large batch size, while masters are trained on a small portion of the batch (which are sampled from the large batch size trained on workers) with a high-precision model.

- We investigate quantization methods and design a logarithmic quantization method with two factors. Instead of using full-precision (i.e., 32-bit floating points) representation, we restrict the values of parameters on workers to be either powers of two or zero. To minimize the error caused by quantization, a re-scaling strategy called *bit centering* [26] is integrated in our algorithm. In this way, the error of quantization will converge to zero asymptotically.

- We explore approaches to reduce the variance in training data points and we extend C-LPT to adopt importance sampling for variance reduction. In C-LPT, importance sampling happens only when sampling a subset of the training batch on workers to be trained on the master side. This particular batch on the worker side is uniformly sampled from the whole dataset.

- We conduct extensive experiments using C-LPT with various neural network architectures and real datasets. The results demonstrate that C-LPT can benefit DNN training in two folds. Firstly, the communication overhead is largely reduced as the bi-directional partial gradient updates between masters and workers are both in low-bits. Secondly, the noises introduced from the quantization and the variance of data points in a batch are both addressed elegantly as masters and workers are working in a cooperating manner to compensate for the loss of each other.

1.4.1 Publication Out of This Thesis

One paper has been published from the work reported in this thesis.

The work on real-time hypergraph partitioning in Chapter 3, has been published in: Jiang, W., Qi, J., Yu, J., Huang, J., and Zhang, R. (2018). HyperX: A scalable hypergraph framework. IEEE Transactions on Knowledge and Data Engineering.

1.5 Thesis Outline

The remainder of the thesis is organized as follows.

- Chapter 2 presents a literature review on three related areas: graph and hypergraph partitioning, deep learning and neural networks, and quantized neural network training. As related work in the area of graph partitioning and hypergraph partitioning is quite rich, we only discuss the most related ones.

- Chapter 3 elaborates our proposed real-time hypergraph partitioning algorithm, i.e., streaming refinement partitioning (SRP). We first describe the computation model of HyperX and semi-supervised learning via label propagation based on batch information. We then investigate streaming computation models under different partitioning objectives, and we propose streaming refinement partitioning (SRP). We also report the performance of SRP in empirical studies.

- Chapter 4 elaborates our quantized neural network training paradigm, i.e., cooperated low precision training (C-LPT). We first investigate the bottlenecks and key influencer to the slow training speed and convergence rate problems in deep neural network training. Next, we analyze the effect of different precision level of parameters and gradients to the training accuracy. Then, we describe the three key novel designs in our proposed method, i.e., gradient quantization, gradient aggregation with various precision, and training batch sampling. Last, we compare C-LPT with other quantization methods to evaluate its performance.

- Chapter 5 concludes the thesis. It further discusses limitations of the proposed techniques and suggests possible future work to extend the studies.

Chapter 2

Literature Review

The thesis studies advanced data modeling for efficient distributed computing that considers the scenario of streaming hypergraph partitioning and deep neural network training. Finding an optimal solution to either graph partitioning or hypergraph partitioning is known to be NP-hard when we take communication cost and workload balance constraints into consideration. As a result, a range of heuristics have been proposed to produce a near-optimal solution. We investigate the studies related to this area in Section 2.1. On the other hand, deep neural networks are essential in deep learning to solve machine learning tasks. A lot of neural network architectures have been proposed to achieve high prediction accuracy, which are surveyed in Section 2.2. Recently, to make the training phase more efficiently, compression based methods are developed. We describe previous work on these methods in Section 2.3.

2.1 Graph and Hypergraph Partitioning

Graph partitioning is a well-studied problem and efficient heuristic algorithms have been proposed. Hypergraphs generalize graphs. A hypergraph can be transformed to a bipartite graph, therefore there is a strong connection in graph partitioning and hypergraph partitioning. Some studies try to solve the hypergraph partitioning problems through adopting graph partitioning algorithms. Before we review the work on direct hypergraph partitioning, let us first look at graph partitioning.

2.1.1 Graph Partitioning

Classic graph partitioning studies focus on the minimum bisection optimization. This partitioning goal minimizes the number of edge cuts when partitioning the vertices into two disjoint sets. Normally it is required that each set achieves equal number of vertices. In real applications, partitioning the graph into an arbitrary number of sets is required. The generalized version of the minimum bisection problem is defined as the (k, v)-balanced partitioning problem for k sets and each set with at most v times of the average number of vertices. Therefore, the bisection can be written as $(2, 1)$-balanced partitioning. The graph partitioning algorithms in solving minimum bisection problem can be divided into several categories.

Exact Algorithms. Most exact algorithms rely on the branch-and-bound technique [76]. Bounds are derived using various approaches. [6] uses semi-definite programming and [97] follows constructing multi-commodity flows to retrieve the bounds. Linear programming is used by [7], and a continuous quadratic program is developed by [44].The objective of the quadratic program is decomposed into convex and concave components, which is tackled afterwards by a relaxation. No matter which methods are used, a bottleneck is reached during the partitioning. Either the bounds derived yields small branch-and-bound trees but hard to calculate, or the bounds are weaker and the trees are larger but easier to compute when combined bounds are used. All of these methods typically are used to solve small problems because of their expensive computing cost.

Spectral Partitioning. Spectral techniques in splitting a graph into two blocks are still in use nowadays. This technique was firstly proposed by [35] and a sequence of new methods [9, 48] were developed based on it later on. This technique evaluates the eigenvector corresponding to the second smallest eigenvalue of the Laplacian matrix L of the graph in order to estimate the global connectivity information. The second eigenvector can be deducted by Lanczos algorithm [75]. However, this method is extremely slow when running on modern graph frameworks. In the implementation over HyperX, it is still quite expensive.

Geometric Partitioning. If the coordinates of vertices in a graph is accessible, then they are useful in geometric partitioning. Geometrically grouped regions normally are the subgraphs with low cut cost. A bunch of methods are in this category, such as recursive coordinate bisection [104] and inertial partitioning [39]. A recent work embeds arbitrary graphs into the coordinate space

using a multilevel graph drawing to analysis the geometric information [71].

It is widely accepted that minimum bisection problem is NP-hard. When k is greater than 2, it becomes even harder. So several approximation solutions have been studied. When $v = 2$, a bi-criteria approximation solution [72] achieves $O(\sqrt{\log k \log n})$ approximation ratio;when $v \geq 2$, work [37] achieves an $O(\log n)$ complexity approximation ratio; when $v = 1 + \epsilon$, approximation ratio is $O(\log^2 n)$ [4]. A similar balanced partitioning problem that, unlike (k, v)-balanced partitioning which uses edge-cut, uses vertex-cut as partitioning method. The goal in this problem is to partition the edges into two sets with equal number, while the number of vertices spanning different sets is minimized. Even though the problem is from a different angle, the complexity is still NP hard. To apply efficient partitioning algorithm to the real life applications, heuristics methods are necessary.

2.1.2 Graph Partitioning with Heuristics

In our work, we are specifically interested in Pregel graph processing paradigm. Pregel paradigm is based on bulk synchronous parallel (BSP), and it has been widely adopted in most modern distriubted graph processing frameworks including the one we are working on HyperX. We discuss several practical heuristics that can be implemented in this paradigm. This heuristics often can not provide theoretical guarantee in obtaining the optimal partitioning. However, they are highly efficient and extremely effective when dealing with large scale data records.

There is a multi-level graph partitioning for general graph, called Metis package [64]. It has several partitioning phases: firstly, it coarsens a large graph into smaller ones; then it partitions on the simplified graph with the previously described spectral partitioning algorithm; after that, it uncoarsens the partitioned simplified graphs back to the original large scale graph. When comparing with random partitioning and degree balancing partitioning with different graphs, work [86] shows that the Metis is more sensitive to access patterns in the graph while the other two tend to be more robust.

For problems where the graph size is even large and exceed the memory limits, Metis can not be effectively employed. To tackle the problem, a label propagation algorithm is proposed [115] to partition a graph of size up to billions edges. This is designed especially for online social network and therefore the label is the partition chosen by vertices, and they are initialized according to the

geographic information of vertices. The label propagation runs in an iterative manner. In each iteration, each vertex refers to the label of the other vertices connected to it and deciding whether to migrate with neighbouring vertices or not.

2.1.3 Streaming Graph Partitioning

When it comes to streaming graph partitioning, because of the continuity in the arriving of vertices and insufficient information about the graph at a given time, traditional graph partitioning algorithms tend to be unable to assign the vertices into an ideal partition. So they need to be applied many times to reassign the vertices based on newly coming structure. To be more efficient, direct streaming partitioning algorithms are developed. Fennel [114] is proposed as a one-pass streaming partitioning algorithm which can partition the data streaming with high efficiency on a cluster of workers. Another work [89] extends the Fennel to be able to partition the streaming in a more general way so that the multiple attributes of the vertices are balanced as well. Both algorithms run in an iterative manner.

In the streaming context, the incoming information not only includes newly arrived, unassigned data records, but also may have modifications to the partitioned ones. This happens when the structure of graph tends to change rapidly, for example, friendship relations graph on social networks. In order to tackle this situation, a number of graph partitioning algorithms that can update the partitions efficiently have been proposed. The connectivity-based decentralized node clustering scheme [93] detects the community among the graph locally without requiring for global knowledge. It updates the partitioning according to the evolution of the graph using a scalable algorithm.

Streaming partitioning is largely different from batch partitioning as in the real-time processing requirement. Graph algorithms may run on the partitions while the streaming is still on. This asks the partitioning algorithms not only to consider the graph topological structure but also to estimate the runtime workloads of the partitioned ones and to monitor the historic performance metrics and access patterns. Several dynamic partitioning algorithms fall in this category [53].

LogGP LogGP [123] firstly generates a hypergraph based on the historical access pattern data records, and afterwards provides initial partitioning results using a streaming hypergraph partitioning algorithm. From these initial partitions, LogGP performs a series of mining based

dynamic adjustments based on the topological changes of the graph.

Sedge Sedge [126] is a management system that provides two types of partitioning, static partitioning and dynamic runtime partitioning. The static one is based on graph structures and the dynamic one is based on graph workloads. Different partitioning results from different algorithms are intensively monitored such that the best vertex partitioning is chosen as the final one.

Hama Based on Apache Hama [1], a method [100] is proposed to evaluate the graph workloads in a sliding window so that the vertex partitioning can be optimized during the iterations.

Mizan Mizan [69] performs as a load-balancer for Pregel. It can migrate the vertices between partitions to minimize the communication and computation overhead.

2.1.4 Hypergraph Partitioning

Hypergraph partitioning, as a generalization of graph partitioning, is an even more complex problem. Hypergraph partitioning is previously explored in the context of integrated circuit design (VLSI) with a minimum cut-size objective on hyperedges in order to minimize bisections on a printed circuit board. Exact partitioning algorithms are expensive in both computation and storage space usage. A well known exact partitioning algorithm is the *spectral clustering* which is for partitioning bipartites (note that bipartites and hypergraphs are equivalent). It has been shown that a real-value relaxation under the cut criterion leads to the eigen-decomposition of a positive semidefinite matrix [22]. This means that cuts based on the second eigenvector always gives a guaranteed approximation to the optimal cut. A series of techniques based on spectural clustering have been proposed [34, 113, 129]. However, these techniques are inefficient as the size of a bipartite converted from a hypergraph can be very large. Two popular methods to compute eigen-decomposition are Lanczos [101] and SVD [129]. They both have the time complexity of $O(k(N_x + N_y)^{3/2})$, where N_x and N_y are the number of vertices in each group, respectively. For large hypergraphs where the numbers of vertices and hyperedges are up to 100 millions, spectral clustering may not have satisfactory efficiency.

A parallel version of *spectral clustering* is proposed and evaluated lately [19] and bipartite partitioning is studied among distributed systems such as Hadoop as well [17, 18]. The graphs with billions of vertices are of interest in both work. These work propose the Aweto algorithm

[1] Referent for Apache Hama can be found: https://hama.apache.org/

in which it argues that different types of vertices should be distinguished, and therefore should be partitioned respectively. Three steps are included in this algorithm. Firstly, it partitions over a single type of vertices randomly. Next, minimizing the number of replications in edge-cuts is set up as the objection when partitioning the other types of vertices. Finally, vertices are migrated locally following this work [13] such that the local score is maximized.

Heuristic based partitioning algorithms such as hMeTis [66], PaToH [14], Parkway [112], and Zoltan [33] have been developed for a higher partitioning efficiency. The algorithms hMetis and PaToH are single-machine based algorithms, while the rest of the algorithms can run in a distributed manner. All these algorithms share the same multi-level coarsen-uncoarsen technique to partition a hypergraph. This technique coarsens the original hypergraph to a sequence of smaller ones. Then, heuristic partitioning algorithms are applied to the smallest hypergraph. Finally, the partitioned hypergraph is uncoarsened back to produce partitions of the original hypergraph. These algorithms require random accesses to the hypergraph located either in the memory or in other nodes. Thus, they do not scale well. Furthermore, these algorithms use MPI APIs and cannot be easily reimplemented on parallel frameworks such as Spark. Another technique called *hMulti-phase refinement* [98] considers hypergraph partitioning as a global optimization problem but it shares the same limitations. There are more recent tools for hypergraph partitioning. UMPa [32] is a serial partitioner that aims at minimizing several objective functions simultaneously; rFM [99] allows relocating vertices in partitioning; HyperSwap [127] partitions hyperedges rather than vertices.

2.2 Deep Learning and Neural Networks

In this section, we describe the development of deep learning and discuss popular architectures of modern neural networks that we used for empirical studies in Chapter 4.

2.2.1 Deep Learning

Deep learning is a branch in machine learning. It solves machine learning tasks using deep neural networks (DNNs), which consist of a collection of neurons. A neuron performs a form of math-

ematical function in the way of a simulation to a biological neuron in the brain. It can receive as many as inputs from neurons in previous layers, then generate one output using a linear combination of weighted sum of the inputs. To represent more complex mathematical functions, this output need to pass an *activation function* in order to get the final output to next layer. *Activation function* is usually non-linear. The *activation function* $f(x)$ can be any non-linear functions. We list some as following:

- **Sigmoid**: Sigmoid is a continuous and differentiable function that is able to map a real number into the interval of $[0, 1]$. It is represented as:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.1}$$

- **Hyperbolic Tangent (tanh)**: This function maps a real number to the interval of $[-1, 1]$. It is continuous and differentiable as well. It is calculated following:

$$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{2.2}$$

- **Rectified Linear Unit (ReLU)**: This is a rectifier function that is continuous but is not differentiable at zero. It is given by:

$$f(x) = \begin{cases} 0, & when \;\; x \leq 0 \\ x, & otherwise \end{cases} \tag{2.3}$$

Neurons are organized as layers. Neurons in the same layer do not connect to each other. Neurons with no previous layers are called *inputs*, and neurons with no next layers are called *outputs*. The layers between inputs and outputs are called *hidden layers*. The number of hidden layers can be more than one. Figure 2.1 shows a neural networks with one hidden layer. When the number of *hidden layers* is relatively large, for instance greater than eight, it is considered as a "deep" neural network [74]. Modern deep neural networks these days can have more than hundreds of layers [47]. Each connection between two neurons in neighbouring layers is assigned a weight w. This weight can be adjusted when different inputs are send into the networks. This adjust-

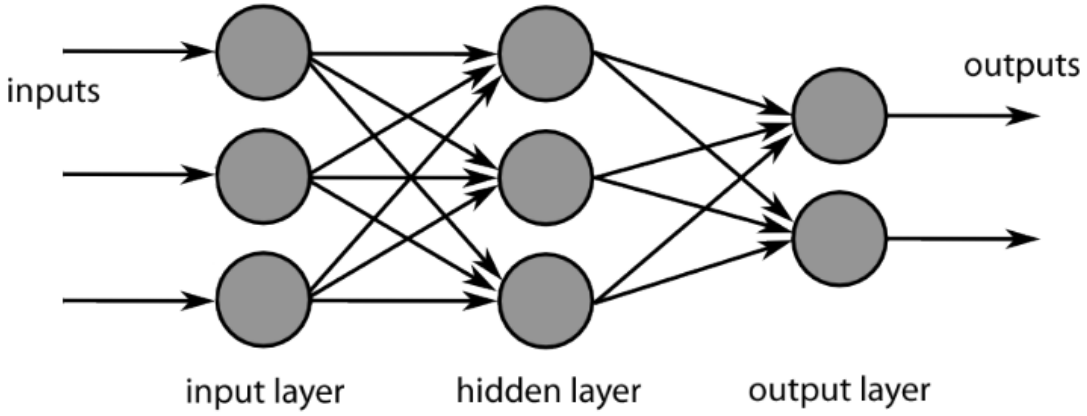


Figure 2.1: A neural network with one hidden layer

ment typically use gradient descent technique. The procedure of updating the weights according to different inputs is called *network training*.

Gradient descent is a first-order numerical optimization method in finding the local optimal by calculating the gradient of the loss function and moving weights in the negative direction of the gradients with a specific step length. The step length is proportional to the absolute value of the gradient and this ratio is known as *learning rate*. Back propagation (BP) algorithm [77] is the most important part in gradient descent as it is the key step in calculating the gradient. BP consists of four critical steps:

1. **Feed-forward pass (inference)**: The linear combination and non-linear activation functions are evaluated layer by layer from input neuron to the output neuron. The final outputs from output neurons could be continuous value when the problem is regression or discrete values if the problem is a classification problem. This outputs could be right or wrong. It is decided by using a certain *loss function*.

2. **BP on output layer**: Firstly, an error value is calculated by this loss function. Next, the derivatives are calculated on loss function using this error value. At last, the derivatives are propagated back to the previous layer as gradients.

3. **BP on hidden layers**: The gradients are calculated iteratively following the fashion of previous step until it reaches the input layer.

4. **Weight updates**: Each weights are updated using the corresponding gradients at the same place.

A cycle of above four steps is called a training iteration. Iterations continue until the BP algorithm converges (the gradients become sufficiently small).

Deep learning has been applied to a wide range of applications such as image recognition and natural language processing. Beyond image recognition, computer vision domain enjoys remarkable performance enhancement after using deep neural network to generate features automatically [74] instead of using human selected features. A bunch of advanced techniques, such as dropout [106], batch normalization [60], and residual [47], are developed recently. With these techniques adopted, the accuracy in image classification on deep neural networks over large dataset beats the accuracy of human beings [47] for the first time. Many vision based applications, e.g. self-driving vehicles [15] and medical treatments [135], largely benefited from the development of DNNs. Recurrent neural networks (RNNs), as a variant of DNNs, is different from previously described convolutional neural networks (CNNs). RNNs suits the sequential data best. They have the power to greatly improve the accuracy in speech recognition [49], natural language processing [23], and machine translation [8]. Another branch of the applications of DNNs is deep reinforcement learning. This branch is extremely useful in robotic grasping [79] and game playing through self-learning [102].

2.2.2 Neural Network Architectures

We describe the neural network architectures used in the evaluation of our proposed C-LPT paradigm. These architectures include two types of neural networks, i.e., multi-layer perceptron (MLP) and convolutional neural network (CNN).

MLP MLP is constructed with a number of fully-connected layers. Two neighbouring fully-connected layers connect to each other with the help of a non-linear layer in the middle. It is called fully-connected because each neuron in layer i connects all neurons in layer $i+1$ and so on. Therefore, the computation in MLP is basically matrix or vector multiplication. LeNet-300-100 [78] is designed as a MLP with two hidden layers of size 300 neurons and 100 neurons,

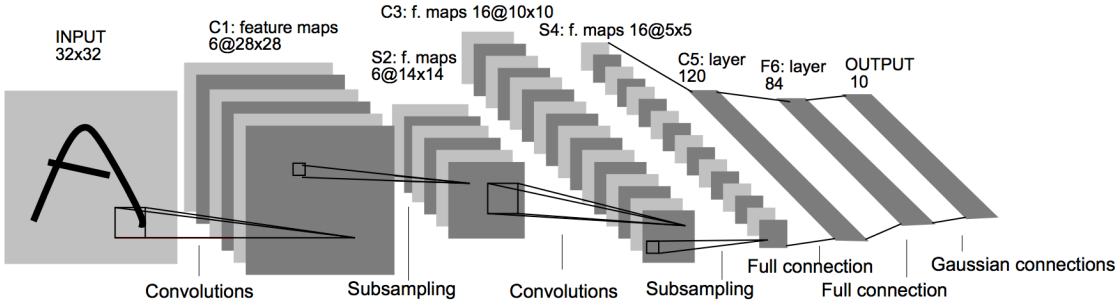


Figure 2.2: LeNet-5 Architecture [78]

respectively.

CNN Being different from MLP, CNN is not fully-connected. This is typically used in image processing. Because of the spatial locality of images, CNN shares weights in different space. And this weight sharing technique makes the model size much smaller compared to using fully-connected layers when the input image dimensions are the same. We evaluate with following four CNNs.

- *LeNet-5* [78] is a CNN with two convolutional layers and two fully-connected layers after them. This relatively simple network is used to recoginze hand written digits. The structure is shown in Figure 2.2.

- *AlexNet* [74] was proposed in 2012. It takes the advantage of strong computation power of GPUs and achieves a remarkable low error rate in image classification, top-1 error rate of 42.8% and top-5 error rate of 19.7% on ImageNet dataset. Before AlexNet, the features are normally manually selected. In comparison, AlexNet lets the CNN train these features on its own instead. The convolutional layers are of different kernel sizes. It starts from size 11×11 and reduces to 5×5 as the layer gets deeper, reaches 3×3 at last. There are 5 convolutional layers and three fully-connected layers, and there are around 60 million parameters in total. Figure 2.3 shows the structure of AlexNet.

- *VGG-16* [105] After the appear of AlexNet, there is a rush in manufacturing deeper CNNs in the several following years. VGG-16 appeared in 2014. It has thirteen convolutional layers and three fully-connected layers with more than 130 million parameters in total. It significantly reduces the error rate even further, top-1 error rate of 21.5% and top-5 error rate

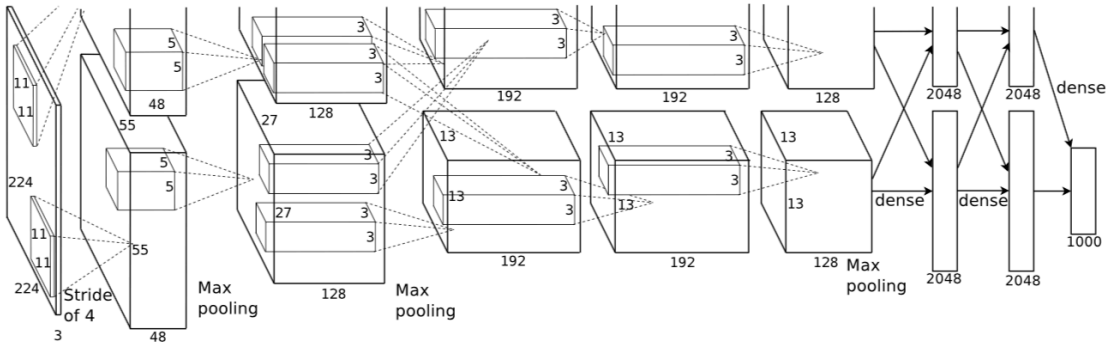


Figure 2.3: AlexNet Architecture [74]

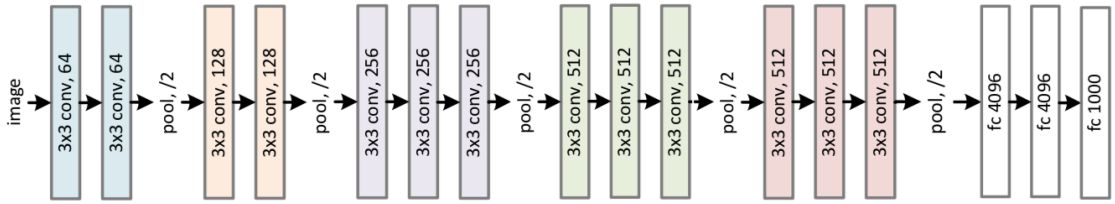


Figure 2.4: VGG-16 Architecture [105]

of 11.3% using ImageNet dataset. The most different part from AlexNet is that VGG uses the same kernal size of 3×3 throughout all convolutional layers. It demonstrates a high generalization capability in many applications. With the development of transfer learning, the pre-trained VGG-16 net using ImageNet has been widely used in areas of image classification, object detection and image segmentation. Figure 2.4 shows the architecture of VGG-16.

- *ResNet* [47] ResNet is another milestone in model manufacturing. It was proposed in 2015, and introduces "bypass" layers to the convolutional parts. These "bypass" layers separate the CNN with several residual blocks. This design is motivated by the observation that the gradients tend to be too small to be able to pass from the output to the input as the number of layers goes higher. The proposal of residual blocks allow gradients to pass more easily from the end to the beginning. ResNet has 49 convolution layers and one fully-connected layer. Therefore it is known as ResNet-50 as well. It has around 26 million parameters to be trained. Each residual block aggregates the current feature map and feature map passes from previous residual block element-wisely. It achieves top-1 error rate of 23.9% and top-5

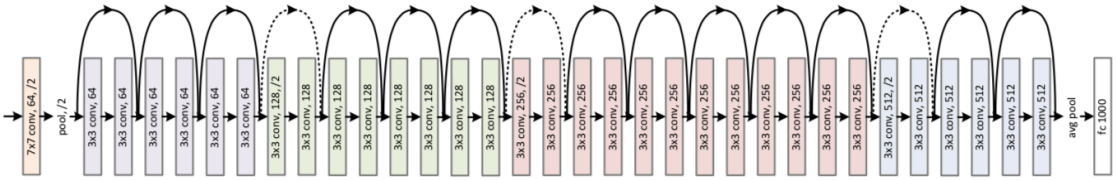


Figure 2.5: ResNet Architecture [47]

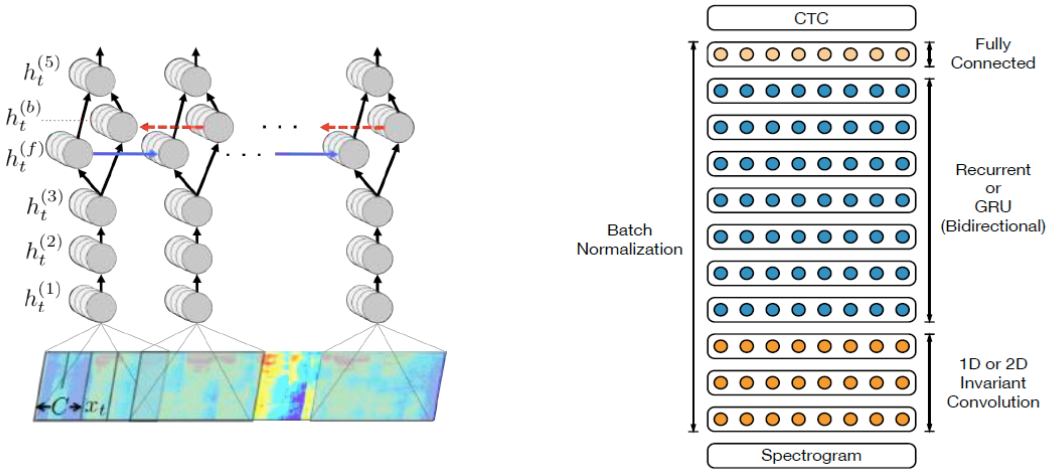


Figure 2.6: DeepSpeech Architecture, DeepSpeech 1 on the left [46] and DeepSpeech 2 on the right [2]

error rate of 7.1% on ImageNet. ResNet-50 makes a perfect balance between computational complexity and accuracy performance. Its structure is shown in Figure 2.5.

- *DeepSpeech* [2, 46] DeepSpeech 1 is a bidirectional recurrent neural network designed for speech recognition task, which has 8 million parameters in total, including five layers of neurons and one bi-directional recurrent layer. DeepSpeech 2 makes improvement on the first version and has a much bigger network in terms of the number of parameters. It has around 70 million parameters with seven bi-directional recurrent layer. The development of DeepSpeech model makes it become the default end-to-end training method for speech recognition instead of previous hybrid NN-HMM model. The illustration of DeepSpeech structure is shown in Figure 2.6.

2.3 Quantized Neural Network Training

There is a large number of studies that try to improve the computational performance of neural networks due to their popularity and importance. A branch of such studies tries to simplify the models while maintaining the accuracy because it is believed that these neural networks with millions of parameters have significant redundancy, which makes the models over-parameterized. Too much redundancy can lead to more computational effort and higher communication costs. Also when deploying them for applications, it is a waste of the memory. Generally speaking, there are two main streams in making networks more concise: pruning the weights and using lower precision (quantize weights to fewer bits). Because in this thesis we put more attention to training neural networks using lower precision, we restrict the scope of the related work to the second stream, quantized neural network training, and describe the existing work.

There have been several work proposing approaches to reduce the precision and bit width. Gong et al. [41] and Wu et al. [122] applied k-means scalar quantization to the parameter values. Vanhoucke et al. [117] explored a fixed-point implementation with 8-bit integer (vs 32-bit floating point) activations and showed that 8-bit quantization of the parameters can result in significant speed-up with minimal loss of accuracy. Abwer et al. [5] used L2 error minimization to quantize the neural networks. Hash function is used in work [20] to build HashedNets so that the bit width of parameters are reduced by grouping weights into hash tables. Hwang et al. [58] proposed an optimization method for neural network with ternary weights and 3-bit activation functions.

More aggressive approaches pushed the bit width even narrower. In the extreme case of 1-bit representation of each weight, we have existing work such as binary weight [24], or ternary weights [137]. Except from quantizing the weights only, work [94, 136] tried to quantize the activation to a low precision as well. Matrix multiplication can be replaced by XNOR if both weights and activation are quantized. Another is to replace the matrix multiplication by shifts, which is even cheaper. This is realized by Miyashita et al. [88]. They adopted logarithmic quantization into the design of the model and experiments demonstrate its competitive performance.

The accuracy of such quantized networks do not show much accuracy loss using small models but the accuracy is significantly lowered when dealing with large CNNs such as GoogleNet. To address this issue, the work in [51] used second-order methods and proposed a proximal Newton algorithm with diagonal Hessian approximation that directly minimizes the loss with respect to the quantized weights. The work in [83] reduced the time on float point multiplication in the

training stage by stochastically binarizing weights and converting multiplications in the hidden state computation to sign changes.

Chapter 3

Real-time Hypergraph Partitioning for Distributed Learning [1]

3.1 Introduction

Among many applications, hypergraphs are used to represent the data records with rich structures and high-dimensional relationships. In these applications, the data records are represented by vertices, and the relationships among records are represented as hyperedges. When such applications involve huge amount of data, the size of the hypergraphs built can be very large. Minimizing the query cost on such hypergraphs is crucial for the applications. In this scenario, hypergraph partitioning helps to partition the query loads to several workers, which enables horizontal scaling of the large-scale hypergraphs.

Hypergraph partitioning requires a decomposition of a full hypergraph into multiple subsets such that the inter-dependency between sets is lower than the intra-dependency between the elements in the same subset. This applies to a variety of applications in practice, in which one need to partition a set of items into disjoint components such that similar items are assigned to the same component in order to minimize the number of relationships between groups. This problem arises in the context of clustering of information objects such as documents, images and web-pages. For example, the goal may be to partition given collection of documents into sub-collections so that the maximum number of distinct topics in each sub-collection is minimized.

Partitioning hypergraphs in these applications require a clear trade-off between data locality and workload balance, as the intra-dependency and inter-dependency correspond to local and re-

[1] Part of this chapter has been published in: Jiang, W., Qi, J., Yu, J., Huang, J., and Zhang, R. (2018). HyperX: A scalable hypergraph framework. IEEE Transactions on Knowledge and Data Engineering.

mote accesses to the data records, respectively. Minimizing the inter-dependencies will lead to a partitioning result with higher data locality, and therefore results in a speed up in distributed processing as less communication is needed between workers. When the size of hypergraphs reaches billion-level, the data to be communicated can easily go up to gigabytes, which causes saturated networks. So the less communication, the higher efficiency we obtain. This locality benefits large scale hypergraph processing as the network latency between workers is much higher than the latency between processors. However, minimizing the inter-dependencies may cause unbalanced workload. This is because the size of sub-connected components generally do not equal to each other. If a worker has more data to process than the others, then the distributed system has to wait for this worker while the others are idle. This is why, in most of the cases, the performance of a distributed system is limited by the response of the slowest worker. This trade-off between data locality and workload balance posts a great challenge to hypergraph partitioning.

A number of hypergraph partitioning algorithms have been proposed to solve this problem, including spectral partitioning [31], bisection graph growing [65], and max-flow min-cut approach [3]. However, these algorithms share very similar drawbacks summarized as follows.

First, these algorithms evaluate the quality of partitioning based on either the number of edge cuts or vertex cuts. This simple optimization goal is reasonable before large distributed computing platforms, such as Pregel [85], Hadoop, and PowerGraph [42], were developed. But it is not suitable to distributed hypergraph processing.

Second, most of these algorithms typically must load and access the entire hypergraph into main memory in order to take the whole hypergraph structure into consideration. Therefore the computational cost of these algorithms can be very intensive. As the size of hypergraphs is growing rapidly, the existing methods become prohibitively expensive. To achieve the best performance in hypergraph processing, designing more efficient hypergraph partitioning algorithms become urgent.

Third, most of these algorithms are built upon hypergraphs in a store-and-process manner where the structure of the whole hypergraph is already known and all other information about hypergraph such as the degrees of vertices either are available or can be calculated very soon. This makes optimization much easier. However, in many recent hypergraph applications, the structure of a hypergraph is constantly changing. In such a scenario, the hypergraph are represented as a dy-

namic hypergraph stream which is composed of a sequence of hyperedge insertions and deletions. The existing partitioning methods that require batch information are unfortunately impossible to apply on partitioning this kind of hypergraphs as only a sub-graph is available to the algorithm at a time.

In this chapter, we focus on real-time hypergraph partitioning. In the real-time partitioning settings, vertices arrive one at a time in a sequential manner. Each should be assigned to one partition in a very short time after it arrives, so for time complexity, we need a near linear time algorithm to partition. This streaming model could be simulated by restricting the batch processing model with limited memory that is allowed to be used at any time during the execution. This limited memory is at most linear to the number of partitioned subsets. This is non-trivial as hypergraph partitioning has been shown to be an NP hard problem.

The remainder of this chapter is organized as follows. We firstly describe the computation model of HyperX and semi-supervised learning via label propagation based on batch information. We investigate streaming computation models under different partitioning objectives, and we propose streaming refinement partitioning (SRP). We also report the performance of SRP in empirical studies.

3.2 Preliminaries

3.2.1 HyperX: Scalable Hypergraph Framework

HyperX has a similar architecture (cf. Fig. 3.1) to an existing graph framework, GraphX [43]: 1) it builds on top of Spark; 2) it runs on the Hadoop platform, i.e., YARN and HDFS; and 3) it shares all the optimization techniques with GraphX. HyperX *directly stores* a hypergraph as two RDDs [128], `vRDD` for the vertices and `hRDD` for the hyperedges.

Computation Model

Algorithms running on hypergraphs usually involve accessing and updating not only $v.val$ but also $h.val$. For example, when mining relationships among social media networks [38], the weight of relations ($h.val$) between visual descriptors ($v.val$) needs to be gradually learned during the

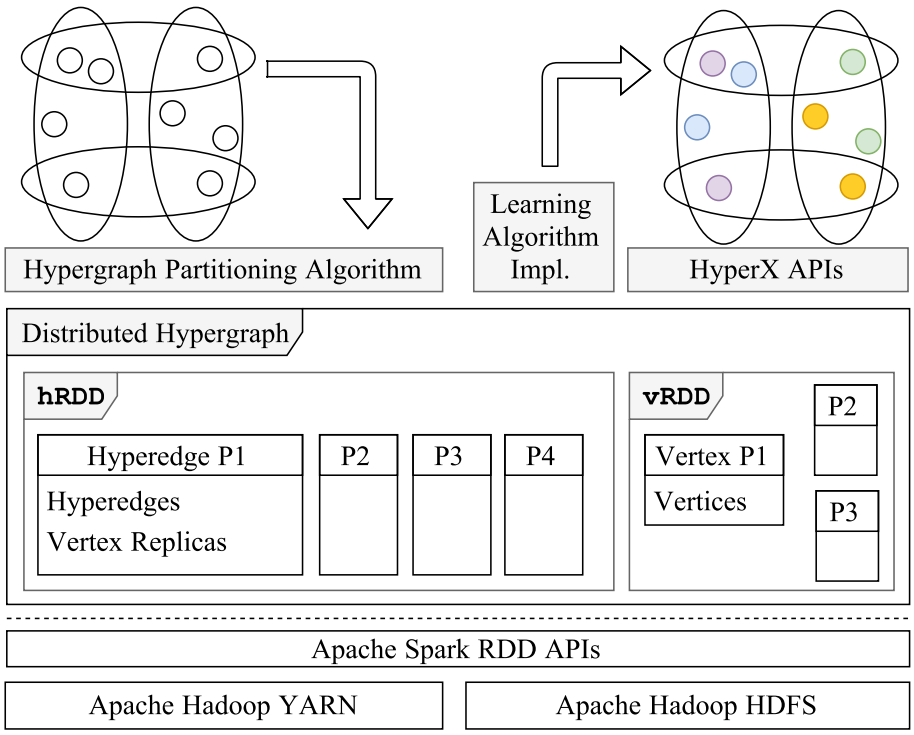


Figure 3.1: HyperX Structure

computation. Thus, HyperX provide both a vertex program `vProg` and a hyperedge program `hProg`. The vertex program runs on each vertex, and computes $v.val$ of the vertex based on all the $h.val$s of the incident hyperedges which have that vertex inside. The hyperedge program runs on every hyperedge to update $h.val$ according to all the $v.val$s of its incident vertices.

As illustrated in Fig. 3.2, to update $h.val$ and $v.val$, HyperX takes only one iteration, while SE takes two iterations. Meanwhile, having `hProg` makes it much easier to balance the workloads in the two steps during each iteration because in the first step all the vertices participate in `vProg`, and in the second step, all the hyperedges participate in `hProg`. These two steps are fully decoupled. The hyperedge program provides another benefit on efficiency: it avoids extensive communication costs between vertices by adding a local aggregation step on the hyperedge partitions before sending messages to other partitions.

Storage Model

To distribute the workload of hypergraph processing is to separate and assign vertices and hyperedges of a hypergraph to the distributed workers (unit of resources in Spark). This requires a

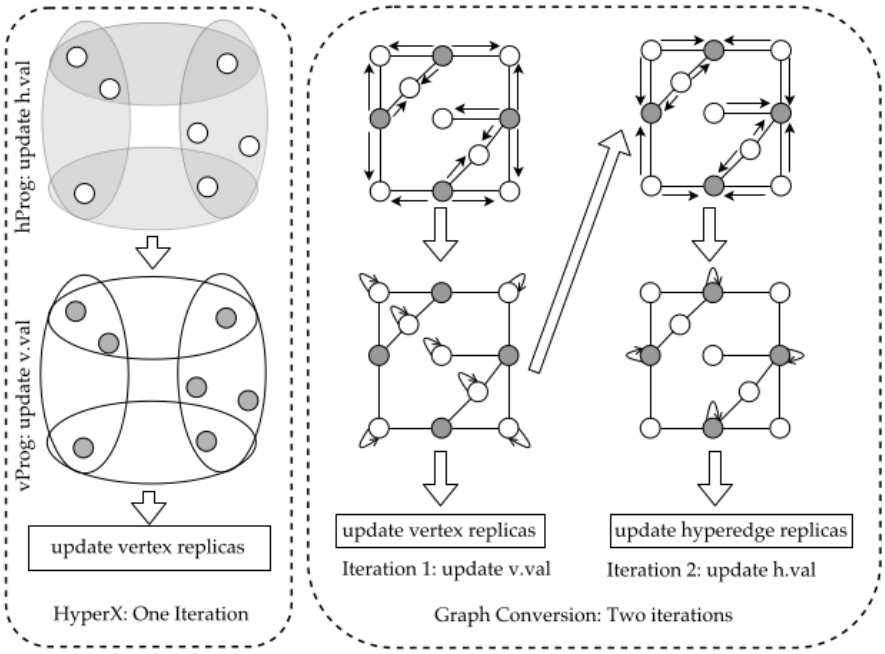


Figure 3.2: Comparing HyperX with Graph Conversion, the gray shapes and bold arrows indicate the running of `vProg` and `hProg` in each step

hybrid-cut that disjointedly separates the vertices *and* the hyperedges. This differs from either the *vertex-cut* that cuts the vertices to disjointedly separate the edges [13] or the *edge-cut* that cuts the edges to disjointedly separate the vertices [72].

Following the convention, in HyperX, the vertices whose incident hyperedges are assigned to different workers are replicated to those workers. Hyperedges are not replicated because replicating hyperedges is prohibitive as each hyperedge connects an unrestricted number of vertices, which need to be replicated with the hyperedge. This helps avoid the excessive replicas observed in CE and SE.

As a result, `vProg` does not operate locally. Instead, hyperedge values are sent to the vertex partitions over the network. The communication cost of `vProg` is thus attribute to the number of vertex replicas, as $h.val$ only needs to be sent to a partition where there are replicas of the vertices in the partition of h. Another network communication cost comes from updating $h.val$ according to the changed $v.val$, which is also attribute to the number of replicas. Both types of communications have been optimized by employing local aggregation, which means that we combine the values destined to the same partition together into one package before sending it out. Thus, HyperX avoids extensive communications between the partitions.

Representing Hypergraphs on HyperX

HyperX stores a hypergraph as one `vRDD` and one `hRDD`. Conceptually, each vertex and each hyperedge is stored as one row in its corresponding RDD. Let `vid` and `hid` denote the id of a vertex and a hyperedge, respectively. While `vRDD` simply stores $(\texttt{vid}, v.val)$ pairs, `hRDD` deals with an arbitrary number of vertices in each hyperedge. Directly storing a vertex set in one row introduces an overhead of the object headers and the linking pointers. Instead, HyperX flatten each hyperedge by storing every vertex of it as a tuple of three values $\langle \texttt{vid, hid, isSrc} \rangle$, where `vid` is the id of the vertex, `hid` is the id of the hyperedge, and `isSrc` is a boolean denoting whether the vertex is a source vertex for directed hyperedge. For undirected hyperedge, `isSrc` is not used. This enables an efficient (columnar) array implementation. Now each hyperedge may span multiple consecutive rows in the `hRDD`. Given a `hid`, we cannot access the corresponding hyperedge directly. To resolve this, HyperX creates an additional map structure to associate a `hid` with the first row where the hyperedge is stored in the `hRDD`. Compared with the cost of directly storing hyperedges which is attribute to $O(\sum_{h \in \mathcal{H}} a_h)$, the cost of this additional structure is only attribute to $O(n)$, where n is the number of hyperedges.

APIs

HyperX is built upon Apache Spark's RDD abstraction and its corresponding dataflow APIs. Here, an RDD can be seen as a horizontally distributed table. HyperX provides eight major APIs: `vertices`, `hyperedges`, `tuples`, `mrTuples`, `joinV`, `subV` `mapV`, and `mapH`, as listed in Table 3.1. The first three functions provide tabular views of a hypergraph, which are used to read data. The last two functions are setters for $v.val$ and $h.val$, which are used for hypergraph initialization. The middle three functions `mrTuples`, `joinV`, `subV` are essential for hypergraph processing. These core functions enables the implementation of an iterative computation paradigm similar to Pregel [85]. The paradigm is provided as an API in HyperX, named `HyperPregel`.

3.2.2 Semi-supervised Learning via Label Propagation

Label propagation has been widely used in graph classification problems. Algorithm proposed

Table 3.1: The APIs of `Hypergraph[V, H]` in HyperX

Functions	Return	Usage
vertices	`RDD[(Id, V)]`	View
hyperedges	`RDD[(Id, H)]`	View
tuples	`RDD[(Map[Id, srcV], Map[Id, dstV], H)]`	View
mrTuples	`RDD[(Id, M)]`	Update
joinV	`Hypergraph[V2, H]`	Update
subH	`Hypergraph[V, H]`	Update
mapV	`Hypergraph[V2, H]`	Set
mapH	`Hypergraph[V, H2]`	Set

in [139] provides an approach for classification using label propagation on streams. The input is a weighted undirected graph $G = (V, E, w)$, in which there are l labeled points $(x_1, y_1), \ldots, (x_l, y_l)$ and u unlabeled points $x_{l+1}, \ldots, x_{l+u}$, typically $l \ll u$. The weight of an edge (x, y) represents some measure of similarity between two endpoints. Nearby points in Euclidean space are assigned large edge weight. The task is to assign labels to nodes $U = \{l+1, \ldots, l+u\}$ based on labeled nodes $L = \{1, \ldots, l\}$.

The learning method is to first compute a real-valued function $f : V \to \mathbb{R}$ on G. The function f can be separated into two parts $f_u \in \mathbb{R}^U$ and $f_l \in \mathbb{R}^L$ according to the partition $V = U \cup L$. The f_l is given as input, the f_u is the part need to compute. We want unlabeled points that are nearby in the graph to have similar labels. The algorithm computes the function f_u by minimizing the following *energy function* of the graph

$$E(f) = \min_{f_u} \frac{1}{2} \sum_{(x,y) \in E} w_{x,y}(f(x) - f(y))^2 \tag{3.1}$$

This is the same as minimizing $\frac{1}{2} f^T \mathbf{G} f$ under the given function f_l, where $\mathbf{G}$ is the Laplacian of G.

3.3 Partitioning Objective and Theoretical Analysis

3.3.1 Partitioning Objective

Suppose we have k workers. The hypergraph partitioning problem is to allocate m vertices and n hyperedges to the k workers. Let binary variables $x_{h,i}$ and $y_{v,i}$ denote whether a hyperedge h and a vertex v are assigned to the i^{th} worker, where $i \in \{1,2,3,...,k\}$. Then we can get Equations (3.2). A partition result is denoted as $\{X,Y\}$. It is a particular set of values for all the variables $x_{h,i} \in X = \{0,1\}^{n\times k}$ and $y_{v,i} \in Y = \{0,1\}^{m\times k}$.

$$\sum_{i=1}^{k} x_{h,i} = 1, \sum_{i=1}^{k} y_{v,i} = 1, x_{h,i} = 0,1; y_{v,i} = 0,1 \tag{3.2}$$

Given a vertex v, let $N(v)$ denote the set of its incident hyperedges and $R(X,Y_v)$ denote the number of replicas of vertex v given a partition result $\{X,Y\}$. Then

$$R(X,Y_v) = \sum_{i=1}^{k} max((1 - y_{v,i} - \prod_{h\in N(v)} (1 - x_{h,i})), 0) \tag{3.3}$$

This formulation of $R(X,Y_v)$ considers the local aggregation mechanism implemented in popular distributed framework, i.e., on each partition only one vertex replica is necessary no matter how many hyperedges in that partition are incident to the vertex. A vertex will only receive messages from a partition where it has a replica as the replica indicates the presence of incident hyperedges. Meanwhile, when vertex values change, the vertex replicas need to be updated, the communication cost of which is again attribute to the number of replicas. Thus, each replica incurs two units of communication cost. Let $C(X,Y)$ denote the overall communication costs:

$$C(X,Y) = 2 \times \sum_{v\in\mathcal{V}} R(X,Y_v) \tag{3.4}$$

As the space cost is proportional to $R(X,Y)$, i.e., the number of replicas given a partition result $\{X,Y\}$, the minimization of the communication costs minimizes the space cost as well. Thus, the optimization problem is as follows.

$$\begin{aligned} &\textbf{minimize} \quad C(X,Y) \\ &\textbf{subject to} \quad \sum_{h\in\mathcal{H}} x_{h,i} a_h \leq (1+\alpha)\frac{\sum_{h\in\mathcal{H}} a_h}{k} \\ &\qquad\qquad \sum_{v\in\mathcal{V}} y_{v,i} R(X,Y_v) \leq (1+\beta)\frac{\sum_{v\in\mathcal{V}} R(X,Y_v)}{k} \end{aligned} \tag{3.5}$$

Here, α and β are non-negative relaxation factors. A value 0 for these factors suggests that every worker should have exactly the same workload. The inequalities are the load balancing constraints over hyperedges (the input of `hProg` is determined by a_h) and vertices (the input of `vProg` is determined by $R(X, Y_v)$), respectively.

The constrained optimization problem described above involves a trade-off between the communication and space costs and the potential overheads for hypergraph learning algorithms to process the partitions.

Setting proper values for α and β is vital for the optimization. However, the values of α and β may not be determined easily as the trade-off involves multiple factors, e.g., the data distribution, the computation complexity of `hProg` and `vProg`, the network bandwidth, etc. To overcome this limitation, we investigate a soft-constrained variation of the above problem.

3.3.2 The Strict Case

Consider a case where $\beta = +\infty$ and the variables in Y are configured such that every vertex is assigned to a worker that has its incident hyperedges. We optimize a strict instance in this case where $\alpha = 0$, i.e., each worker has exactly the same hyperedge workload. Then the optimization problem becomes:

$$\begin{aligned} &\textbf{minimize} \sum_{v \in \mathcal{V}} \sum_{i=1}^{k} (1 - \prod_{h \in N(v)} (1 - x_{h,i})) \\ &\textbf{subject to} \sum_{h \in \mathcal{H}} x_{h,i} a_h \leq \frac{\sum_{h \in \mathcal{H}} a_h}{k}, i \in \{1, 2, ..., k\}. \end{aligned} \tag{3.6}$$

We have the following Theorem.

Theorem 3.1. *The above minimization problem has no polynomial time solution that can achieve a finite approximation factor unless P=NP.*

The proof follows a reduction from the strongly NP-Complete `3-Partition` problem where the goal is to partition the hyperedges set $\mathcal{H}$ into k workers with equal workload. This suggests that in general the strict case is inapproximable (e.g., no polynomial-time approximation scheme (PTAS), no constant approximation factor, or even $\Omega(2^n)$ approximation factor).

3.3.3 A Variant with Soft Constraints

If we are able to quantitatively compare the cost of communication and computations over hyperedges and vertices, we can convert the hard constraints (Inequalities (3.5)) to soft constraints and integrate them into the optimization objective. This gives an alternative minimization objective $C'(X, Y)$.

$$\begin{aligned} C'(X,Y) &= 2 \times \sum_{v \in \mathcal{V}} R(X, Y_v) \\ &+ w_h (\sum_{i=1}^{k} |\frac{1}{k} \sum_{h \in \mathcal{H}} |h| - \sum_{h \in \mathcal{H}} x_{h,i} |h||^p)^{\frac{1}{p}} \\ &+ w_v (\sum_{i=1}^{k} |\frac{1}{k} \sum_{v \in \mathcal{V}} R(X, Y_v) - \sum_{v \in \mathcal{V}} y_{v,i} R(X, Y_v)|^p)^{\frac{1}{p}} \end{aligned} \tag{3.7}$$

Here, w_h and w_v denote the relative cost of a unit of computation cost on hyperedges and vertices compared with a unit of communication cost, respectively; p is the norm to aggregate the workload difference on the partitions.

To minimize this objective, we can reformulate it as a *generalized constraint satisfaction problem* (GSCP) [92] with three payoff functions C_0, C_1, and C_2 as follows.

$$\begin{aligned} C_0(X, Y_v) &= \prod_{h \in N(v)} (1 - x_{v,i}) + y_{v,i} - 1, \forall v \in \mathcal{V} \\ C_1(X) &= -w_h \frac{|\frac{1}{k} \sum_{h \in \mathcal{H}} a_h - \sum_{h \in \mathcal{H}} x_{h,i} a_h|}{\sum_{h \in \mathcal{H}} a_h}, \forall i \in [1, \ldots, k] \\ C_2(X, Y) &= -w_v \frac{|\frac{1}{k} \sum_{v \in \mathcal{V}} R(X, Y_v) - \sum_{v \in \mathcal{V}} y_{v,i} R(X, Y_v)|}{\sum_{v \in \mathcal{V}} R(X, Y_v)} \end{aligned} \tag{3.8}$$

The minimization of the objective $C'(X, Y)$ is therefore equivalent to maximize the total sum of all the payoff functions C_0, C_1, and C_2. This GCSP can be approached by a general semi-definite programming relaxation [92], which has been proven to deliver the best approximation for the GCSP under the unique game conjecture. According to Krauthgamer et al. [72], this relaxed semi-definite programming problem is computable in a polynomial (super-cubic) time.

3.4 Streaming Refinement Partitioning (SRP)

In real-time hypergraph partitioning, we consider the incoming vertices arrive sequentially one at a time and its corresponding hyperedge. The streaming does not necessarily have one vertex at a time but we can always decompose a group of vertices into one of each in sequential to reduce granularity. This streaming computation model allows for limited memory to be used at any time during the execution. The memory is at most linear to the number of components. These assumptions arise as part of requirement for deployment of online services. The vertex and hyperedge are denoted by their *id*s. After the arrival, the vertex should be assigned to a certain partition immediately. If the hyperedge is new, it also needs to be assigned.

Different from batch partitioning, real-time partitioning often requires a fast initial allocation of vertices and hyperedges. This prevents the incoming stream from being blocked. However, this immediate partitioning without a sufficient computation could lead to severe imbalances. In our study, to overcome the imbalances, we relax this requirement so that the initial allocations are allowed to be re-partitioned afterwards. We partition the hypergraph in two runs. This gives us opportunities to focus on achieving fast partitioning in the first run while more balanced partitions in the second run.

In this section, we propose streaming refinement partitioning (SRP) as a variant of the traditional label propagation partitioning (LPP) in streaming scenario. SRP contains two steps.

1. **Rough Partitioning**: In the first step, we use greedy strategy to partition the incoming vertices. This strategy places each incoming vertex onto the partition where the marginal cost of a pre-defined objective function is minimized. It is efficient as it enjoys a linear time complexity. And we use a heuristic *balancing constraint* to avoid severe imbalance.

2. **Iterative Refinement**: In the second step, we put our attention on refining the load-balance. We iteratively refine the partitioning we have obtained from the first step in order to minimize the cost of distributed computing based on our choice of metrics, for example the total number of replicas and the load-balance constraints. We perform a re-partitioning of the hypergraph by running an iterative refinement using label propagation algorithm. We introduce a sliding window on the streaming to limit the number of vertices the partitioning algorithm is running on so that the time and memory constraints in real-time processing is

complied.

3.4.1 Rough Partitioning

Rough partitioning aims to partition the incoming vertex to a partition immediately. The immediate goal is not balancing, but to store the vertex and be ready to accept the next vertex. Different heuristics can be used in the first run to partition the incoming vertex roughly. We describe two of them. In empirical study, we use *greedy partitioning* to evaluate the performance of SRP.

Random partitioning. Random partitioning is a competent contender for initial partitioning due to its high efficiency. It randomly assigns vertices and hyperedges to the partitions. If the number of partitions is small comparing with the data records to be partitioned, a round-robin style random partition may produce almost perfectly balanced partitions [11]. However, it may suffer from arbitrarily high network communication and replication costs, since no hypergraph topology is retained during the partitioning.

Greedy partitioning. Greedy partitioning is a approach to improve random partitioning. This strategy places each incoming vertex onto the partition where the marginal cost of a pre-defined objective function is minimized. This greedy approach has a linear time complexity. Let $\mathcal{H}_i$ denote the set of hyperedges assigned to partition i, and $\mathcal{A}_i$ denote the sum of the arity of the hyperedges assigned to partition i. Then we have

$$\mathcal{H}_i = \{h | x_{h,i} = 1\} \tag{3.9}$$

$$\mathcal{A}_i = \sum_{\{h|x_{h,i}=1\}} a_h \tag{3.10}$$

Let $\langle \mathcal{H}_i \rangle$ be the set of vertices contained in the hyperedges in $\mathcal{H}_i$, i.e., $\langle \mathcal{H}_i \rangle = \{v | v \in h, h \in \mathcal{H}_i\}$. Then the marginal cost of allocating a hyperedge h to the i^{th} worker is

$$M(h,i) = |\langle \mathcal{H}_i \rangle \cup \{v \in h\}| \times c_v + (\mathcal{A}_i + a_h) \times c_h \tag{3.11}$$

where c_v and c_h is the unit cost of balancing the workload of vertex program and hyperedge program, respectively. Note that all these costs are relative values because they are adequate for

determining which workers to assign the hyperedges. After the hyperedges have been allocated, the vertex is similarly partitioned to the i^{th} partition given by

$$P(v,i) = \arg\min_{i}(1 - \prod_{h \in \mathcal{H}_i} x_{h,i}) \times c_r \tag{3.12}$$

where c_r is a relative cost of the communication cost. The pseudo-code of greedy partitioning is listed in Algorithm 1.

Algorithm 1 Greedy Partitioning

Input: Hypergraph $H = (V, E)$, vertex v received one at a time, partition number k, capacity c of each partition
Output: A partition of $H = (V, E)$ into k parts
Set initial partitions $P_1, P_2, \ldots, P_k$ to empty
while incoming vertex v **do**
Receive vertex v and its hypergraph h
$I \leftarrow \{i : |P_i| \leq c\}$ ▷ Partitions not exceeding capacity
$p_h \leftarrow \arg\min_{i \in I} M(h,i)$
Move h to p_h^{th} partition
$p_v \leftarrow \arg\min_i (1 - \prod_{h \in \mathcal{H}_i} x_{h,i})$
Move v to p_v^{th} partition ▷ Vertex v and hyperedge h are assigned to p_v and p_h
end while
return $P_1, P_2, \ldots, P_k$

3.4.2 Iterative Refinement

In the second step, we put our attention on refining the partitions of load-balancing. We iteratively refine the partitions obtained from the first step in order to minimize the cost of distributed computing based on our choice of metrics, for example the total number of replicas and the load-balance constraints. We perform a re-partitioning of the hypergraph by running an iterative refinement using label propagation algorithm.

Label propagation partitioning (LPP) is a batch hypergraph partitioning algorithm to achieve the soft optimization goal with a high efficiency. This algorithm follows a label propagation procedure that labels the hyperedges and the vertices with the partitions they are assigned to. A hyperedge (vertex) iteratively runs two steps: 1) propagating its label to its incident vertices (hyperedges) and 2) updating its labels based on the labels propagated to it. This algorithm differs

from the classic graph label propagation algorithms in that it labels hyperedges in addition to vertices. This is essential because 1) both the hyperedges and the vertices need to be partitioned and 2) for further hypergraph learning on HyperX, both `vProg` and `hProg` need to be balanced. Existing techniques that label either (hyper)edges or vertices are insufficient because labeling only one type of data may obtain workload balance on that type of data but suffer from skewed workloads on the other type of data. By labeling both, LPP guarantees that both vertex and hyperedge workloads are balanced.

In practice, streaming partitioning requires algorithms that run under tight time and memory constraints, that is to be independent of the stream length n. However, the LPP algorithm is a batch algorithm that runs on whole hypergraph. Directly adopting that updating method to streaming processing would result in a growing rate at least as a linear function of the stream length n. This is unacceptable in real-time processing. To overcome the limitation of batch label propagation, we introduce a sliding window of size ϕ on the streaming to limit the number of vertices the partitioning algorithm is running on. That is to say, it maintains a sub-hypergraph H that contains at most ϕ vertices along with their hyperedges. When a new vertex arrives, we add it to H and evict the oldest vertex.

In each iteration of the SRP algorithm, when updating the label for a hyperedge (vertex), possible labels are the partitions that its incident vertices (hyperedges) have been assigned to. To find the optimal one, we first sort all candidates according to a score computed based on our objective function, i.e., balancing the workloads and reducing the replicas, and then choose the one with the highest score. The optimization problem is then reduced to designing two scoring functions $S(h,i)$ and $S(v,i)$ for a hyperedge h and a vertex v, respectively. Specifically, when updating hyperedge label, SRP focuses on minimizing the number of replicas, since it is impossible to compute the arity distribution before all hyperedges are assigned. To reduce the number of replicas, the scoring function $S(h,i)$ for hyperedge h and worker number i is defined as the number of incident vertices of h that choose the worker. Let $N(h)$ denote the incident vertices of h, $L(h)$ and $L(v)$ denote the labels of h and v, respectively. The update rule for a hyperedge is:

$$
\begin{aligned}
L(h) &= \arg\max_{i\in[1,\dots,k]} S(v,i) \\
&= \arg\max_{i\in[1,\dots,k]} |\{v|v \in N(h) \wedge L(v) = i\}|
\end{aligned}
\tag{3.13}
$$

Similarly, when updating the vertex labels, SRP focuses on balancing the hyperedge arity and minimizing the replicas, since it is impossible to compute the replica distribution before all vertices are assigned. To balance the arity, we assign vertices to partitions with smaller sum of arity. Formally, we compute the sum of hyperedge arity for each partition, denoted as $A_i = \sum_{L(h)=i} a_h$, and update vertex v as:

$$
\begin{aligned}
L(v) &= \arg\max_{i\in[1,\dots,k]} S(h,i) \\
&= \arg\max_{i\in[1,\dots,k]} (|\{h|h \in N(v) \wedge L(h) = i\}| \times e^{\frac{\bar{A}^2 - A_i^2}{\bar{A}^2}})
\end{aligned}
\tag{3.14}
$$

Here, $\bar{A} = \frac{\sum_{i\in K} A_i}{k}$, the cardinality $|\cdot|$ accounts for reducing the number of replicas; and the exponent accounts for weighting workers inversely to their sum of arity. The pseudo-code of SRP with sliding window is listed in Algorithm 2.

Discussion. SRP outperforms classic hypergraph partitioning algorithms in the following aspects.

1. SRP is scalable as it is efficient in computing. Unlike coarse-uncoarse techniques, workload on single machine in iterative refinement with SRP is rather small and balanced. The updates of labels rely on message passing.

2. SRP is effective because it partitions the hypergraph streaming in two steps. It considers general constraints on both hyperedges and vertices. It does not replicate hyperedges which avoids high computation and communication costs for processing excessive replicas in hypergraph processing.

3. SRP is efficient because it is designed for streaming partitioning. Its time complexity of each iteration is constant as we use a sliding window to build a sub-hypergraph. This outperforms the state-of-the-art technique [98], in which each subproblem has a super-cubic time complexity and is impractical to large hypergraphs.

Algorithm 2 Label Propagation Partitioning With Sliding Window ϕ

Input: Hypergraph $H = (V, E)$ based on sliding window ϕ, partition number k
Output: A partition of $H = (V, E)$ into k parts
Let initial partitions $P_1, P_2, \ldots, P_k$ be the current partitioning result sets
while $L(v_0)$ is stable **do** ▷ $L(v_0)$ is the label of first vertex in the sliding window
$v_n \leftarrow$ streamloader(H) ▷ Loader reads new vertex from results of rough partitioning
$V \leftarrow \{V - \{v_0\}\} \cup \{v_n\}$ ▷ Keep the size of sliding window
Rebuild sub-hypergraph $H = (V, E)$ ▷ On receiving new vertex, update H
for $j \leftarrow 1$ to $Iter$ **do**
for $h \in E$ **do**
$L(h) \leftarrow \arg\max_i S(h, i)$ ▷ Update the label of hyperedges in E
$A_{L(h)} \leftarrow A_{L(h)} + a_h$ ▷ Calculate the arity
end for
for $v \in V$ **do**
$L(v) \leftarrow \arg\max_{i \in K} S(v, i)$ ▷ Update the label of vertices in V
end for
for $i \leftarrow i$ to k **do**
$A_i \leftarrow 0$ ▷ Reset the arity of each partition
end for
end for
end while

SRP is designed for a general-purpose hypergraph processing framework. Its computational paradigm in the second step that iteratively updates labels of vertices and hyperedges that using the *vertex program* and the *hyperedge program* is commonly supported by graph frameworks such as GraphX [43] and PowerGraph [42]. This enables SRP to be implemented over any platform as long as appropriate data structures can be used to support the hypergraph representation. This requires an *object* that holds the attribute of a hyperedge along with the attributes of vertices involved in this hyperedge so that the label of a hyperedge can be updated according to the labels of its incident vertices. Particularly, HyperX provides a **`tuples`** object to view hyperedge and its vertices as a whole. In this sense, SRP fits HyperX very well. We therefore implement SRP over HyperX to evaluate its performance in the experimental study.

3.5 Experiments

In this section, we evaluate the performance of our proposed streaming refinement partitioning (SRP) approach. We implement SRP on HyperX [2].

3.5.1 Experimental Settings

We firstly describe the experimental settings for the evaluation, including datasets, evaluation metrics and comparative approaches. The experiments are carried out on an 8 virtual-node cluster created from an academic computing cloud [3] running on OpenStack. Each virtual-node has 4 cores running at 2.6GHz and with 16GB memory. Note that each worker corresponds to one core. A single node running with 4 processes effectively simulates 4 workers. The network bandwidth is up to 600Mbps. One node acts as the master and the other 7 nodes act as slaves (i.e., up to $w = 28$ workers) using Apache Hadoop 2.4.0 with Yarn as the resource manager. The execution engine is Apache Spark 1.1.0-SNAPSHOT. HyperX is implemented in Scala.

Datasets

Three real datasets are used, Medline Coauthor (Med) [4], Orkut Communities (Ork) and Friendster Communities (Fri) [125]. These datasets are publicly available on the web. We also use synthetic datasets. These synthetic datasets are generated using Zipfian distribution with exponent $s = 2$. The details about dataset used in our experiments are listed in Table 3.3, where c_{vd} and c_{va} are the coefficient of variance of the vertex degree and the hyperedge arity, respectively. We transformed these datasets into undirected hypergraphs and eliminated loop circles from the original lease. Interconnections between vertices are rather intense in hypergraphs represented by these datasets, e.g., the CE graph of Ork and Fri datasets contain 122 billion and 1.8 billion edges, respectively, as shown in Table 3.2. This is of similar magnitude to the size of datasets used in recent studies such as [43].

[2] Source code of SRP is publicly available on github at: `https://github.com/Jedshady/SRP_HyperX`.
[3] Nectar: https://nectar.org.au/
[4] SBNS datasets: http://www.autonlab.org/autonweb/17433.html

Table 3.2: Comparison on the size of datasets

Representation	Ork		Fri	
	$\|H\|$	$\|V\|$	$\|H\|$	$\|V\|$
HyperX	2,322,299	15,301,901	7,944,949	1,620,991
GraphX-CE	2,322,299	122,956,922,990	7,944,949	1,806,067,135
GraphX-SE	17,624,200	1,086,434,971	9,565,940	643,540,869

Evaluation Metrics

We use following metrics to evaluate the partition result of our proposed algorithm SRP.

Partitioning time. Due to the stream processing, the partitioning time includes two time costs. The first one is vertex allocation time. It counts the average time duration from submitting a hypergraph vertex till the completion of partitioning. The other one is total running time which represents the total partitioning time on whole graph. The total running time is not equal to average time duration times number of vertices because the second step in SRP is for refinement. It typically requires longer time than the first step, rough partitioning.

Cut size. For hypergraph partitioning, *cut size* plays an important role. Cut size reflects the communication cost among partitions. It is related to the total replicas of all vertices. As we mentioned in section 3.3.1, the replica number has a direct influence on the communication cost of the partitioned hypergraph. This is because a modification to a vertex should be broadcasted to all its replicas. This indicates the number of cuts on hyperedges between partitions in hypergraphs. Alternatively, we could use edge cut percentage. It is defined as

$$\text{edge cut percentage} = \frac{\text{cut size}}{|H|} \tag{3.15}$$

where $|H|$ is the total number of hyperedges. This is the basic metric to evaluate the equality of partitioning results.

Performance on hypergraph learning algorithms. We run three hypergraph learning algorithms on the partition result of SRP to reflect the partitioning quality indirectly. These learning algorithms are described as following:

- *Random Walk (RW).* On a hypergraph, *random walks* (RW) rank unlabeled data with regard to their high-order relationships with the labeled data. Random walks are carried out on a hypergraph by iteratively executing the following two steps independently: 1) compute the

Table 3.3: Datasets presented in the empirical study

Dataset	$\lvert\mathcal{H}\rvert$	$\lvert\mathcal{V}\rvert$	d_{min}	d_{max}	$\bar{d}$	σ_d	c_{vd}	a_{min}	a_{max}	$\bar{a}$	σ_a	c_{va}
Medline Coauthor (Med)	3,228,002	8,007,214	1	5913	10	36.91	3.69	2	744	4	2.15	0.54
Orkut Communities (Ork)[125]	2,322,299	15,301,901	1	2958	46	80.23	1.74	2	9,120	71	70.81	1.00
Friendster Communities (Fri)[125]	7,944,949	1,620,991	1	1700	5	5.14	1.03	2	9,299	81	81.39	1.00
Synthetic (Zipfian $s = 2$)	2,000,000	8,000,000	2	803	32	33.7	1.05	2	48,744	8	178.59	22.32
		12,000,000	5	1,173	48	50.27	1.05	2	49,526	8	174.07	21.76
		16,000,000	10	1,527	63	66.56	1.06	2	49,006	8	171.36	21.42
		20,000,000	15	1,893	79	83.40	1.06	2	49,963	8	175.52	21.94
		24,000,000	21	2,305	95	100.00	1.05	2	49,326	8	173.12	21.64
	4,000,000	16,000,000	1	1,102	32	36.04	1.13	2	49,843	8	173.12	21.64
	5,999,984		1	940	21	25.04	1.19	2	49,728	8	179.55	22.44
	7,999,535		1	799	16	19.42	1.21	2	49,526	8	173.84	21.73
	9,996,355		1	716	13	15.79	1.21	2	49,932	8	173.84	21.73

stationary probability on each vertex by aggregating the incident hyperedge values and 2) aggregate the probabilities from incident vertices to update the value on each hyperedge.

- *Label Propagation (LP).* As described in section 3.2.2, LP is a semi-supervised learning algorithm. When running on a hypergraph, it finds communities among the vertices according to the high-order relationships among them. The procedure is straightforward: each vertex is assigned a label to start and then iteratively exchange its label with neighboring vertices through hyperedges. In each iteration, every vertex updates its label to a new one by a majority vote from its neighbors. The procedure is similar to RW, except that now $h.val$ and $v.val$ are the labels instead of the stationary probabilities, and there is no starting vertices.

- *Spectral Learning (SP).* SP covers a wide range of hypergraph Laplacian based clustering and semi-supervised learning techniques. Given a hypergraph $\mathcal{G}$, let D denote its vertex degree diagonal matrix, H denote its vertex-hyperedge incident matrix, A denote its hyperedge arity diagonal matrix, and W denote its hyperedge weight diagonal matrix. Then the normalized Laplacian is: $\mathcal{L}_{\mathcal{G}} = I - D^{-1/2}HWA^{-1}H^{T}D^{-1/2}$, where I is an $m \times m$ identity matrix. Straightforwardly, $\mathcal{L}_{\mathcal{G}}$ can be obtained by multiplying the matrices. In HyperX, $\mathcal{L}_{\mathcal{G}}$ can be computed via HyperX APIs with simpler computation, leveraging the fact that diagonal matrix multiplication is simply scaling the corresponding entries. This avoids the expensive matrix multiplication. The algorithm consists of two subtasks: 1) computing the Laplacian matrix, and 2) eigen-decomposing the matrix. We employ the Lanzcos method [101]. On HyperX, Laplacian matrix can be implicitly computed during the matrix-vector multiplication phase. The idea is that a matrix-vector multiplication is basically a series of multiply-and-add operations, which can be decomposed and plugged into the Laplacian computation. The multiplication can be realized using `mrTuples` while the addition is simply `mapV`.

Baseline Methods on Online Partitioning

We compare the partitioning result of the first step using greedy strategy with other online algorithms:

- *Random.* This randomly assigns each vertex to a partition

- *Balance Big.* This treats the incoming vertices in two different ways depending on the degree of vertices. It assigns vertices with large degree to the least loaded partition and vertices with small degree based on greedy. A vertex with degree greater than 100 is considered large.

We compare the final partitioning result after the completion of the second refinement step with several offline techniques, such as Aweto [18], hMetis [66], Zoltan [33], and Parkway [112].

3.5.2 Partitioning Time of Partitioning Algorithms

To compare the vertex allocation time, we compare the partitioning time of greedy strategy with other online partitioning methods, i.e., random, balance big, on all the real datasets. The vertex average partitioning time is listed in Table 3.4. The streaming rate is simulated as shown in the table. It is measured in vertices per second. We use different scale of streaming rate to simulate the real-world situations. Among three algorithms, random partitioning algorithm processes the incoming vertex in the fastest way, which followed by greedy then balance degree. The average time of vertex allocation is shown in millisecond. A reasonable amount of processing time in the streaming setting is that the processing time less than or equal to the streaming rate. When the system can not process the data at the speed data arrive, the length of the waiting queue will keep increasing. Therefore, it is less applicable to the real-time settings. Compared to the streaming rate we tested, it demonstrates that all three heuristic rough partitioning algorithms are reasonably suitable for streaming partitioning as the average partitioning time is smaller than streaming rate. We will evaluate the performance of their partitioning results in the following two sections.

To compare the total partitioning time on whole graph, we compare the partitioning time of SRP with offline partitioning methods, i.e., hMetis, Zoltan, on all the real datasets. The results are listed in Table 3.5 (hMetis5 and hMetis1 are similar and hence only hMetis5 is shown), where k denotes the number of workers in use. Because we are comparing with offline algorithms, we do not set streaming rate in particular. The rate in this part can be considered as file reading in speed. Note that the hMetis and Zoltan implementations are serial, written in C, and highly optimized, while SRP is written in Scala and runs on multiple layers: HyperX, Spark, Hadoop, and JVM. Even assuming that a distributed hMetis implementation can speed up in a (unlikely) linear manner, i.e., dividing the time by $k = 28$, SRP is still faster than both hMetis and Zoltan as

Table 3.4: Partitioning time with online algorithms

Dataset	Algorithm	Average Time t (ms)	k	Streaming Rate (v/s)
Med	**Greedy**	**450**	28	1.9×10^{-2}
	Random	113	28	
	Balance Big	1627	28	
Ork	**Greedy**	**674**	28	8.5×10^{-1}
	Random	97	28	
	Balance Big	2475	28	
Fri	**Greedy**	**460**	28	6.7×10^{-3}
	Random	84	28	
	Balance Big	1870	28	

Table 3.5: Partitioning time with offline algorithms

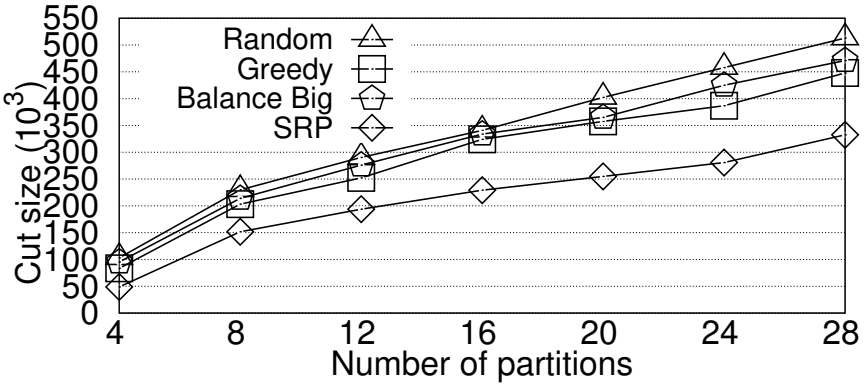
Dataset	Algorithm	Total Partition Time t (s)	k	w.r.t. SRP
Med	**SRP**	**582**	28	**1.0**
	hMetis5	14,796	1	1.1
	Zoltan	4,764	28	8.7
Ork	**SRP**	**947**	28	**1.0**
	hMetis5	88,936	1	4.2
	Zoltan	9,180	28	10.3
Fri	**SRP**	**460**	28	**1.0**
	hMetis5	6,766	1	0.8
	Zoltan	2,875	28	6.9

shown in the last column of Table 3.5.

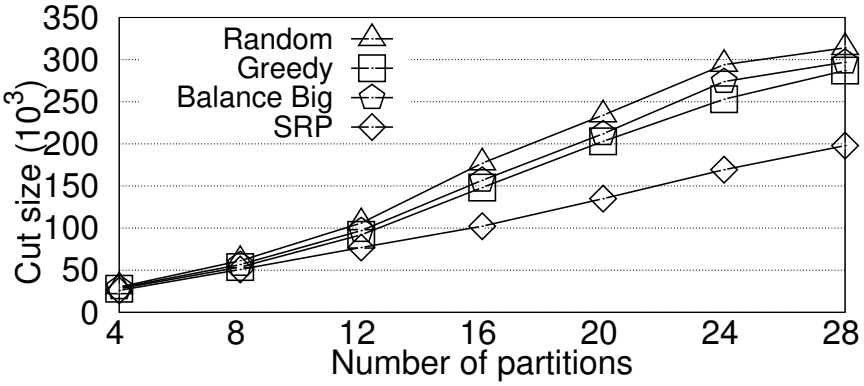
3.5.3 Cut Size of Partitioning Algorithms

In this section, we measure the *cut-size* of the partitions from two angles. Firstly, we evaluate the reduction in *cut-size* between the step of rough partitioning and the step of refinement to demonstrate the necessity of the second partitioning step. Then we compare the *cut-size* of our proposed algorithms SRP with other offline algorithms.

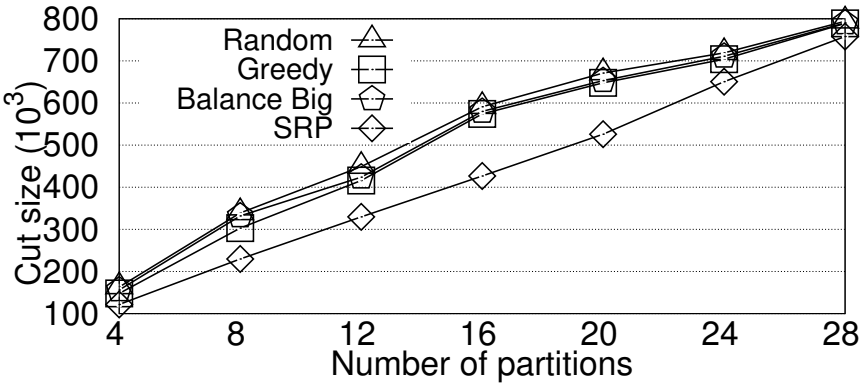
Figure 3.3 shows how many *cut size* is reduced by running the refinement. *Cut size* can be considered as replication factor. The larger the number, the more replicas of vertices among all partitions. Therefore, there will be more communication cost when running hypergraph learning algorithms. As in Figure 3.3, we evaluated *cut size* for different online algorithms with different



(a) Cut size reduction, Med

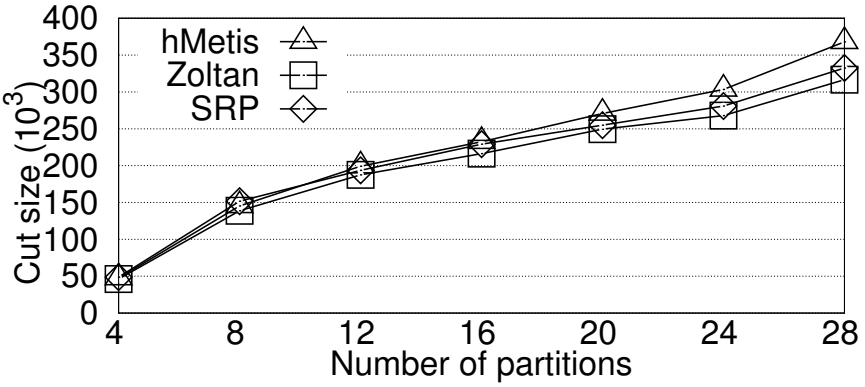


(b) Cut size reduction, Ork

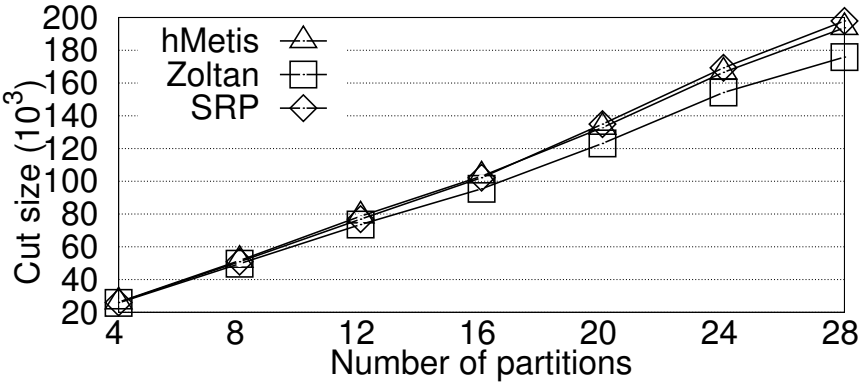


(c) Cut size reduction, Fri

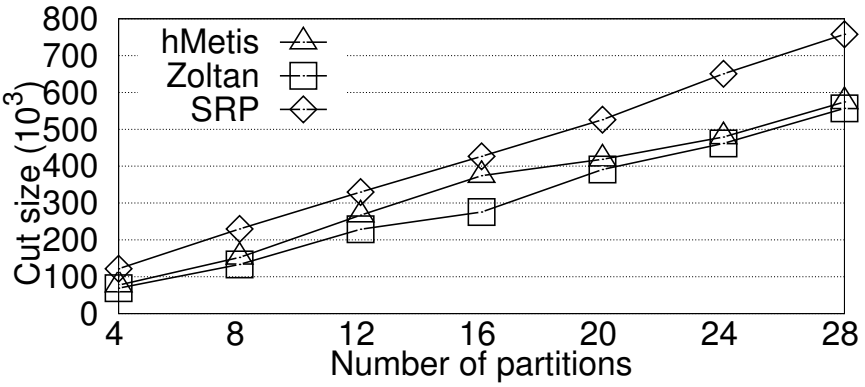
Figure 3.3: Evaluate the cut size reduction from rough partitioning to SRP



(a) Cut size, Med



(b) Cut size, Ork



(c) Cut size, Fri

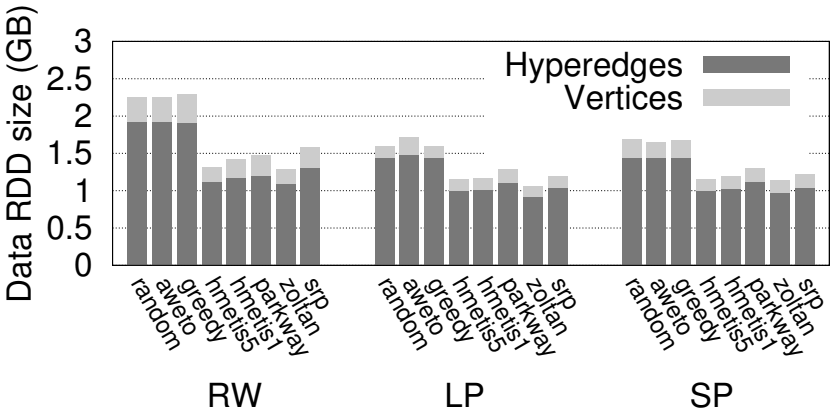
Figure 3.4: Comparing cut size of SRP with offline partitioning algorithms

number of partitions. As the number of partitions increases, the *cut size* increase dramatically as well. This is because, intuitively, if we put the whole hypergraph in one partition, there will be no cuts at all. Comparing Figure 3.3a, 3.3b, and 3.3c, we can see the pattern of increasing varies for different datasets. This is because the distribution of vertices degree and hyperedge arity varies drastically between different datasets. Despite the inconsistency in increasing pattern, the experimental results on three datasets share some similarity. For three online partitioning algorithms, greedy partitioning strategy always slightly outperforms the other two no matter how many partitions are partitioned into. This complies with the conclusion of previous studies. So we use greed as default strategy in rough partitioning. After running refinement, the *cut size* of SRP is smaller than rough partitioning by at least 20% on Med and Ork. On Fri, this number is around 7% because the inherent structure of Fri is not easy to optimize. It has way more hyperedges than vertices, which always results in a cut no matter how to allocate the vertices. Overall, this indicates the refinement partitioning by using label propagation can significantly improve the quality of partitioning in terms of the decrease in *cut size*.

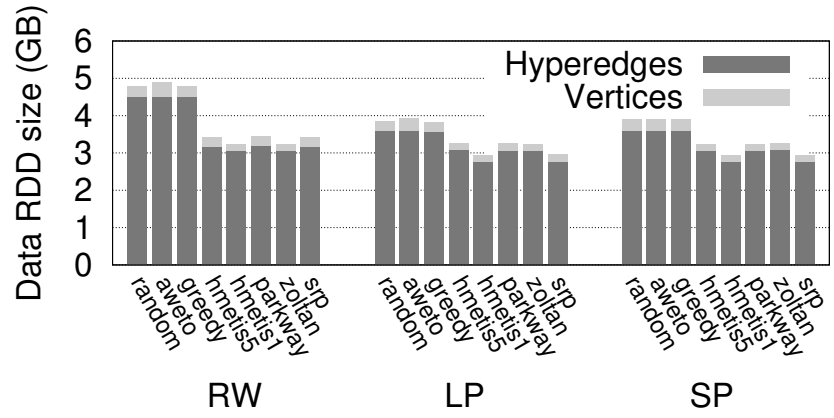
We compare the *cut size* of SRP with other offline partitioning algorithms. The result is shown in Figure 3.4. Note that while hMetis, Parkway, and Zoltan algorithms minimize the cut-size as their optimization goal, our SRP algorithm does not optimize the cut-size explicitly. However, as shown in Fig. 3.4, the cut size of SRP is not much worse than those of hMetis, Parkway, and Zoltan, while SRP shows a much better performance for the other quality measures as discussed above. We also find that Random, Greedy, and Aweto generally produce higher cut-size than those of SRP. Parkway and Zoltan produce very similar results. For simplicity, we only discuss Zoltan as well as hMetis and the proposed algorithm SRP in Fig. 3.4. Zoltan has the minimum cut-size over the three datasets. SRP has around 10-20% higher cut-size compared with those of Zoltan as the number of partitions grows larger, while the cut-size of hMetis lies in between those of SRP and Zoltan.

3.5.4 Quality of Partitioning of Hypergraph Learning Algorithms

We compare the partitioning quality indirectly by evaluating the performance of hypergraph learning algorithms running on the partitions obtained. For hMetis, we set the workload balance factor to 5 and 1. For Zoltan and Parkway, we set the imbalance factor $\epsilon = 0.05$. For SRP, we exe-



(a) Space cost, Med



(b) Space cost, Ork

Figure 3.5: Comparing the space cost of partitioning algorithms

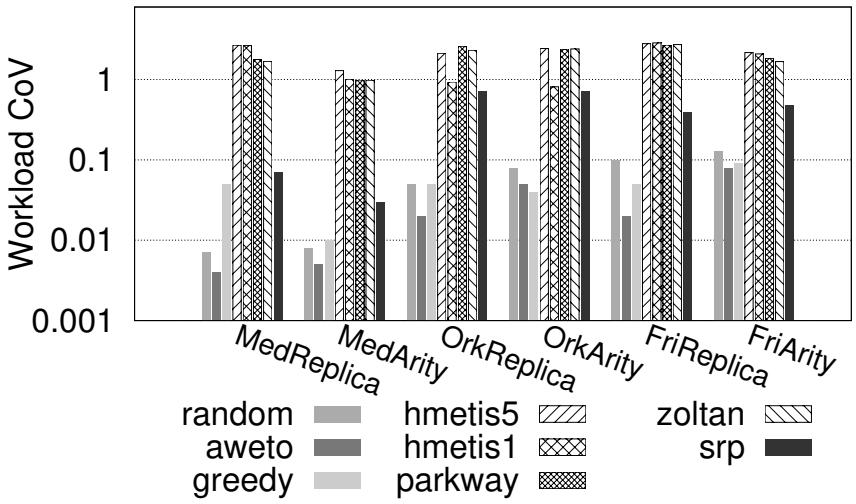
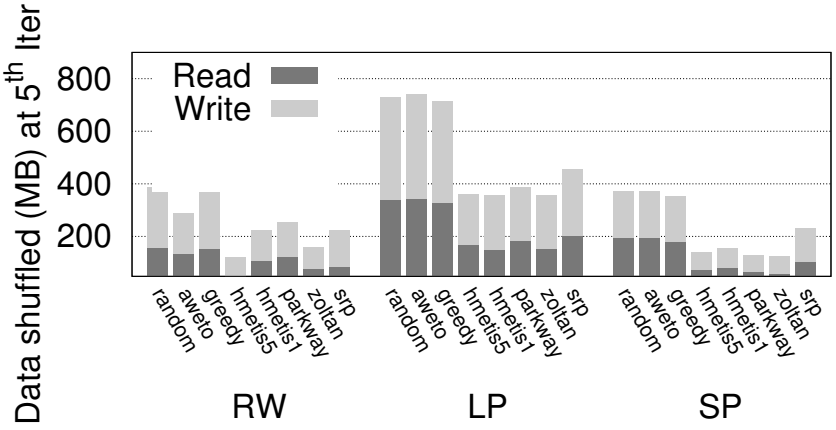
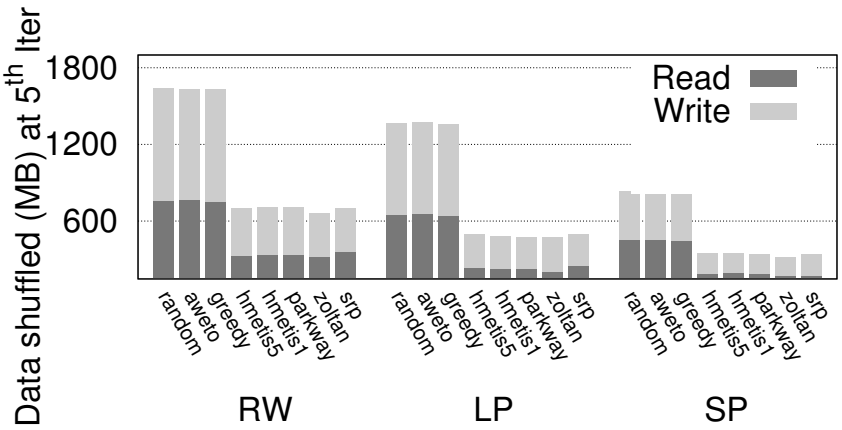


Figure 3.6: Comparing the workload balance



(a) Communication cost, Med



(b) Communication cost, Ork

Figure 3.7: Comparing the communication cost of partitioning algorithms

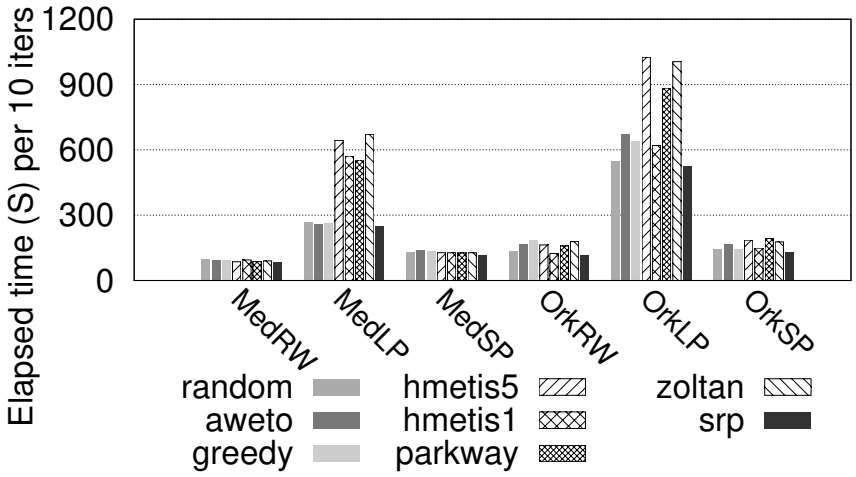


Figure 3.8: Comparing the elapsed time

cute `HyperPregel` for 10 iterations on each sliding window because we find 10 iterations are enough to produce high quality partitions and require a low running time. For all the algorithms, we set the number of partitions k to 28, which is determined by worker numbers. We then run the three hypergraph learning algorithms RW, LP and SP as described in earlier on the partitions created by these algorithms. The results of space cost and results of communication cost on Med and Ork are presented in Figure 3.5 and Figure 3.7, respectively. The workload balance is shown in Figure 3.6 and the learning execution time is shown in Figure 3.8. Among all the algorithms, hMetis, Parkway, Zoltan and SRP outperform the others in all the measures. But SRP is even better in achieving balanced workload. The workload on each partition is defined as the sum of the number of hyperedges and the number of vertices (including replicas) on that partition. In terms of workload balance, which is measured by the *Coefficient of variation* (CoV) of the workloads among different partitions. Random, Aweto, and Greedy all perform well in this metric as shown in Figure 3.6. However, their excessive numbers of replicas offset the advantages and result in high costs in the space and communication metrics. When comparing hMetis, Parkway, and Zoltan with SRP, we observe that they have different preferences on the trade-off between the workload balance and the number of replicas: SRP achieves more balanced workloads than the best of hMetis (Figure 3.6) even though it produces slightly more replicas (Figures 3.5a,3.5b).

As shown in Figures. 3.5a, 3.5b, 3.7a and 3.7b, the extra replicas in SRP do *not* result in significant space or communication overhead. According to Figure 3.8, SRP always outperforms hMetis, Parkway, and Zoltan and delivers up to 2.6 times speed-up for the learning algorithms. Another drawback of them is that they perform particularly poor in LP (even much worse than Random). This is because when all the vertices and hyperedges are active in all iterations, the unbalanced workloads outweigh the benefit gained from a slightly smaller number of replicas.

3.6 Summary

In this chapter, we investigated the real-time hypergraph partitioning problem where vertices arrive one at a time in a sequential manner. We formulated it as an integer programming problem to minimize the number of replicas during the partitioning, therefore minimize the communication costs when running hypergraph applications. We designed a streaming refinement partitioning

(SRP) algorithm which partitions a streaming hypergraph streaming in two steps. In the first step, rough partitioning, we investigated practical heuristics and propose a greedy strategy to create fast and rough partitions. In the second step, iterative refinement, we used label propagation with a fixed size sliding window to make the streaming partitioning algorithm independent of the streaming length in order to comply with the time and memory constraints in real-time processing. We evaluated SRP against a number of online and offline partitioning algorithms with extensive experiments on both real datasets and synthetic datasets. The results demonstrated that SRP is suitable for streaming partitioning as the average partitioning time is smaller than streaming rate. The results showed that SRP can deliver competitive partitioning results compared to that of offline partitioning algorithms and achieve a higher efficiency in the procedure of partitioning.

Chapter 4

Distributed Neural Network Training with Quantization

4.1 Introduction

Hypergraphs have been shown to be highly effective when modeling a wide range of applications where high-order relationships are of interest. Applying deep learning techniques on large scale hypergraphs is challenging due to the size and complex structure of hypergraphs. For machine learning tasks over hypergraphs, studies have shown that using deep neural network (DNN) can improve the learning outcomes. This is because the learning objectives in hypergraph analysis are becoming more complex these days, where features are difficult to define and are highly-correlated. DNNs can be used as a powerful classifier to construct features automatically. Hypergraph analysis using the combination of hypergraphs and DNNs can be found in many applications these days and achieves a remarkable success. For example, when detecting emotions of a person [56], following the workflow shown in Figure 4.1, facial images firstly pass through a convolutional neural network to be decomposed into several hidden expression features; next, high-order relationships between emotional features are depicted by hyperedges for emotion prediction. Another example is to create hypergraph embeddings using DNNs [87, 134]. This is to represent each vertex of a hypergraph in a latent lower dimensional space. After embeddings are created, DNNs can be applied once again to learn the relationships between these vertices from its own embeddings and embeddings of other vertices.

On one hand, such a process takes the advantage of strong representational power of DNNs as an appearance-based classifier; on the other hand, such a process exploits hypergraph representations to gain benefits from its strong capability in capturing high-order relationships. Incorporating

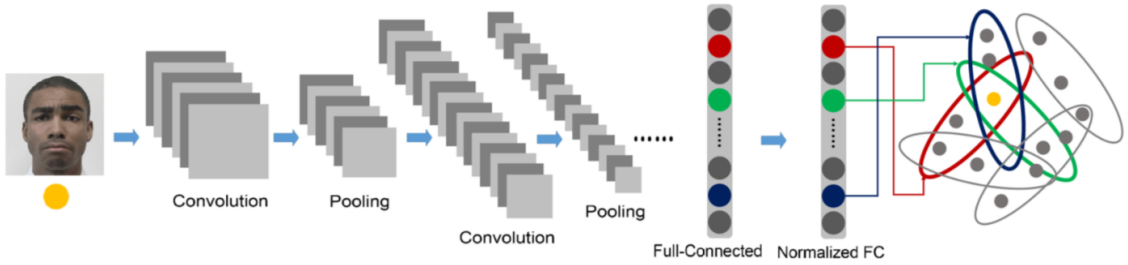


Figure 4.1: Illustration of an application combining hypergraphs and DNNs [56]

deep learning techniques into distributed hypergraph analysis shows a great potential in query processing and knowledge mining on high-dimensional data records where relationships among them are highly correlated.

For distributed hypergraph analysis with deep learning techniques, the performance of the whole work flow depends not only on the hypergraph processing itself, but also on the performance of the DNNs, including phases of training and inference. Training distributed DNNs is known to be difficult to scale due to the inequality between computing time and communication time. Poor scalability will greatly damaged the efficiency of the whole system. In this chapter, we aim to design a DNN training paradigm that is not only suitable for hypergraph analysis, but a more general one that can be applied to many other application scenarios. Distributed hypergraph analysis will definitely benefit from this paradigm.

DNNs have brought substantial advances to a wide range of applications that are driven by large-scale data sets and sophisticated models. Recent work [47, 74, 107, 108] have shown that DNNs can gain benefits on prediction accuracy from increased model depth and width easily. Distributed systems are required in training such models. In the data parallel scheme, models are copied multiple times among the cluster of workers so that they are trained in parallel on different subsets of data that are managed by a data server. Apart from a data server, there is a parameter server that synchronizes the gradients among workers by aggregating all partial gradients that are collected through network communication and broadcasting the new value back. The computation time can be significantly reduced by adding more workers to the cluster. On the other hand, the overhead of gradient synchronization increases dramatically with the growth of the number of workers [81]. The larger the scale of the distributed system is, the more severe the bottleneck of the communication will be. Eventually, this would offset the savings of computing power [80].

Studies propose asynchronous gradient aggregation [50,81,95] to be an alternative approach which gives up on the strictness of the full synchronization among all workers and allows staled gradients that are collected from partial workers being used during the training. This overcomes the communication bottleneck to some extent. However, the staleness of gradients [16, 131] may incur the inconsistency updates of parameters and therefore results in lower prediction accuracy and decelerated convergence rate. Furthermore, the asynchronous methods are troublesome both in implementing and debugging as in its dynamic message flow.

To tackle this communication bottleneck, we investigate the problem from a different angle which keeps the convention of synchronization but considers compressing the model to a smaller size by reducing the precision from 32-bits to a lower number of bits in order to alleviate the burden of communication. Model compression techniques [45,57,83,91,121,136] such as sparse and quantized DNNs have been widely studied for *inference* tasks. The speed and efficiency of *inference* gain huge benefits from the use of modern hardware accelerators such as Google's TPU [63]. However, these accelerators are mainly used in *inference* but not *training* as the influence of reducing precision during *training* has not been fully investigated yet.

Another issue that is orthogonal to the communication bottleneck and that may also impede training efficiency is the inherent variance of stochastic gradient descent (SGD), which causes slow convergence rate. Many variance reduction approaches, such as SVRG [62], SAG [96], SAGA [30], and Importance Sampling [1], have been proposed to remedy this problem. Reducing the variance allows us to use a larger learning rate which leads to a linear convergence rate when optimization function is ideally smooth and strongly convex while vanilla SGD only has a rate of sub-linear.

In this chapter, we study how low-precision and variance reduction can be used to speed up DNN training. We analyze the aforementioned issues and propose a novel training paradigm to train accurate DNNs with lower bits. We called it Cooperated Low-Precision Training (C-LPT).

In C-LPT, we allow masters and workers to keep two different sets of a model in different precision levels. In each training iteration, while the workers train a low-precision model using a large batch size, masters train on a small portion of the batch (which are sampled from the large batch size trained on workers) with a high-precision model. The advantage of such kind of training paradigm contains two folds. Firstly, the communication overhead is largely reduced

as the bi-directional partial gradient updates between masters and workers are both in low-bits. Secondly, the noises introduced from the quantization and the variance of data points in a batch are both addressed elegantly as masters and workers are working in a cooperating manner to compensate for the loss of each other. More specifically, the high-precision model on masters side can make up the loss of precision during the compression and the small portion of data points sampled according to the importance proved to be able to effectively reduce the variance [1]; the large batch size on workers side may potentially provide a broader search of the best gradient descent direction and can significantly minimize the bias.

Another technique we adopted in C-LPT is the compression algorithm. Instead of using full-precision (i.e. 32-bit floating points) representation, we restrict the values of parameters on workers to be either powers of two or zero. To minimize the error caused by quantization, a re-scaling strategy called *bit centering* [26] was integrated in our algorithm. The basic idea lied behind bit centering is that the magnitude of gradients become smaller and smaller when parameters are closer to the local optimum as the training goes. Correspondingly, the power of the low-precision numbers should be dynamically re-scaled because its effective range shrinks. In this way, the error of quantization will converge to zero asymptotically with the converge pace of the algorithm.

The third technique used in C-LPT is importance sampling. Intuitively, training materials are not equally informative and the most informative ones may accelerate the training procedure. In C-LPT, the importance sampling happens only when sampling a subset of the training batch on workers to be trained on the master side. This particular batch on workers side is still uniformly sampled from the whole dataset.

4.2 Preliminaries

4.2.1 Synchronous SGD Parallelization and Communication Overhead

Training tasks using Deep Neural Networks can be written as the following optimization problem. Let $f_1, f_2, \ldots, f_n$ be a sequence of vector functions from $\mathbb{R}^d$ to $\mathbb{R}$. The goal is to find an approximate solution over a finite sum of N components:

$$\text{minimize } P(w), \qquad P(w) = \frac{1}{N}\sum_{i=1}^{N} f_i(w), \quad w \in \mathbb{R}^d \tag{4.1}$$

A standard method is Gradient Descent (GD) using the Back-Propagation Algorithm [77], which can be described by the following update rule, for iterations $t = 1, 2, \ldots, T$, we have:

$$w^{(t)} = w^{(t-1)} - \eta_t \nabla P(w^{(t-1)}) = w^{(t-1)} - \frac{\eta_t}{N}\sum_{i=1}^{N} \nabla f_i(w^{(t-1)}) \tag{4.2}$$

The drawback of GD lies in that it requires a derivative evaluation on each data point in each iteration, which is added up to N times. This is expensive especially on large datasets. An improvement made on GD leads to a much more popular method Stochastic Gradient Descent (SGD), which differs from GD as it randomly samples i_t from $\{1, 2, \ldots, N\}$ at each iteration $t = 1, 2, \ldots, T$, and a further modification is to sample many i_ts. The set of data samples $\{i_t\}$ is called a batch. The batch size is denoted as B. The updating rule of batch SGD is as following:

$$w^{(t)} = w^{(t-1)} - \frac{\eta_t}{B}\sum_{i=1}^{B} \nabla f_{i_t}(w^{(t-1)}) \tag{4.3}$$

The advantage of SGD is that each iteration only relies on a small number of derivative evaluations. Therefore, the computational cost is B/N of the GD. However, the disadvantage it brings in is that the variance introduced by the random sampling. Even though, as pointed out in [62], the expectation $\mathbb{E}[w^{(t)}|w^{(t-1)}]$ in equation 4.3 is identical to that in equation 4.2, the real gradient computed in each iteration can be very different. When difference is large, the large variance may hurt the convergence rate a lot.

It is well known that SGD is very hard to parallelize because of its inherently sequential nature. Figure 4.2 shows conventional structure of the data-parallel SGD [28]. Correspondingly, there is also model-parallel SGD, but we do not discuss it here. Data-parallel SGD is commonly used for single node multi-GPU training and multi-node multi-GPU training. The global batch of B training samples for the current iteration is split into $M * N$ equal sized sets of size b, i.e. $b = B/(M * N)$, where M is the number of workers and N is the number of GPU chips on each worker as shown in Figure 4.2. The gradients g of all workers are then send to masters for aggregation and the updated gradients $\hat{g}$ are then send back to workers to update the model. The whole procedure includes two different levels of synchronization: input synchronization and gradients synchronization.

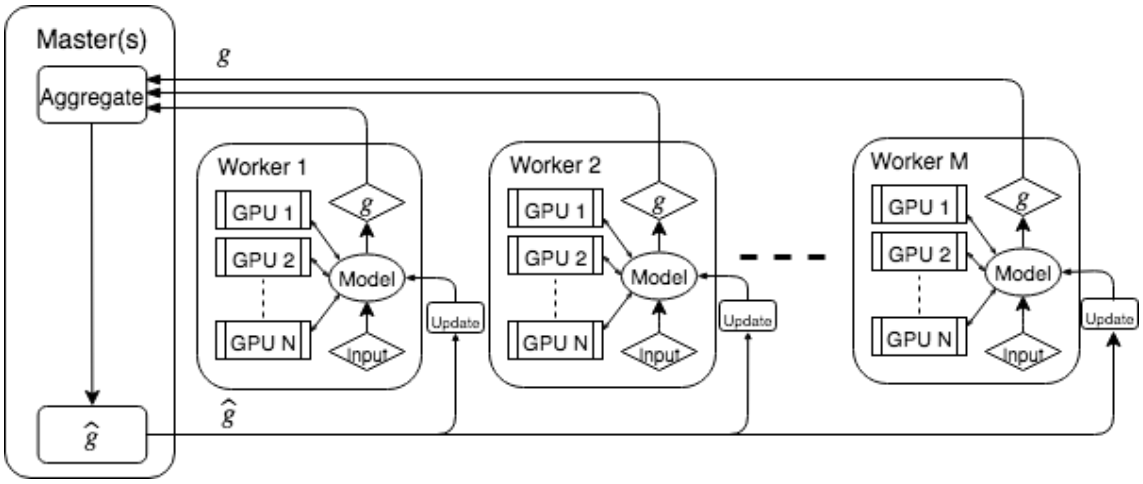


Figure 4.2: Parallel SGD Training.

The gradients synchronization requires the communication of the computed gradients g between masters and all workers in every iteration t. Recently, faster GPUs have largely reduced the computation time which leads to a severe problem that, after scaling to only a few nodes, the communication time exceeds the compute time, leaving the valuable computation units idle. Limited network bandwidth [68] is one of the key bottlenecks to the scalability of distributed DNN training. A quick remedy, which is also a hot research topic, is to compress the gradients g before sending to masters. Figure 4.3 shows the quantized SGD distributed training. Before aggregating the quantized gradients $g_{quantized}$, a decompress step is necessary. One direct influence of the quantization is the reduction of the prediction accuracy. There are many works proposed on exploring quantization methods to overcome this influence. We describe the quantization method we adopted in C-LPT in section 4.3.1.

4.2.2 Importance Sampling

Standard SGD does not incorporate any sampling mechanism as that it does not sample data on purpose in order to maximally reduce the uncertainty. Recently, works such as [1, 119] tried to focus on the most "useful" training data points instead of giving equal attention to all data so that the variance can be minimized during SGD training. Importance sampling is a technique used to reduce variance when estimating an integral of the form:

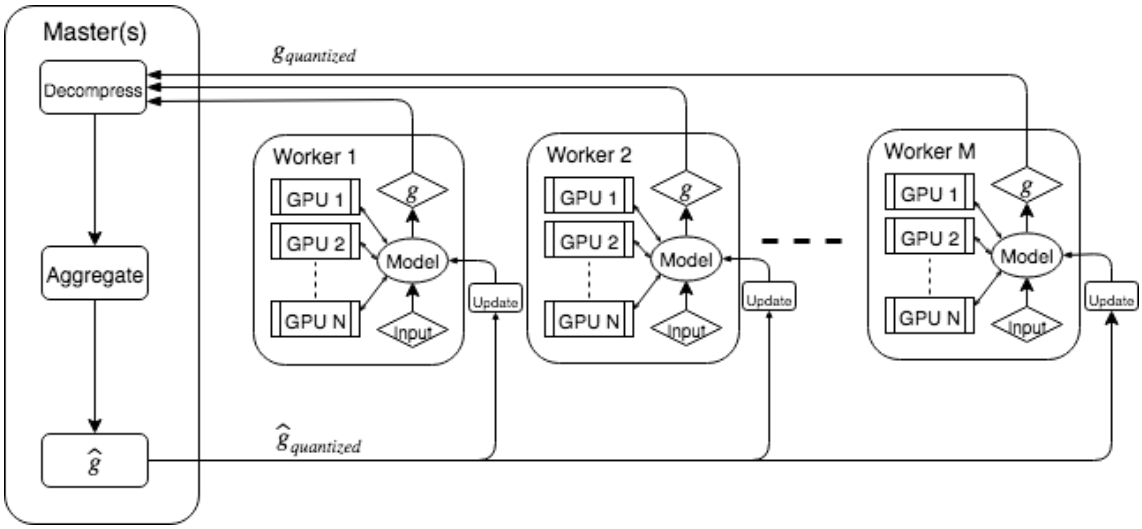


Figure 4.3: Quantized distributed training with multiple GPUs on multiple nodes.

$$\mathbb{E}_{p(x)}[f(x)] = \int p(x)f(x)dx = \mathbb{E}_q[\frac{p(x)}{q(x)}f(x)] \tag{4.4}$$

This integral requires that $q(x)$ is greater than zero no matter when $p(x)$ is greater than zero. The mean of the importance sampling estimator

$$\frac{p(x)}{q(x)}f(x) \qquad while \;\; x \sim q \tag{4.5}$$

is $\mu = \mathbb{E}_p[f(x)]$ so it is unbiased.

Here, x is a random variable in $\mathbb{R}^{d_1}$ and the function $f(x)$ can be any function that maps from $\mathbb{R}^{d_1}$ to $\mathbb{R}^{d_2}$. $p(x)$ is the probability density function of x and $q(x)$ is a valid proposal distribution for important sampling.

To apply above theory to DNNs training, a intuitive choice of objective function is to used the trace of the covariance matrix of the proposal distribution, $Tr(\sum(q))$. This is because the trace is the sum of all the eigenvalues of $\sum(q)$, which in this case is a positive semi-definite matrix. When taking a closer look at this trace, it is also the sum of all the variances for each individual component of the gradient vector.

Then the trace of $\sum(q)$ is minimized by the following optimal proposal q^*:

$$q^*(x) = \frac{1}{K}p(x)\|f(x)\|_2 \qquad while \;\; K = \int p(x)\|f(x)\|_2 dx \tag{4.6}$$

And it achieves the optimal value when

$$Tr(\sum(q^*)) = (\mathbb{E}_p[\|f(x)\|_2])^2 - \|\mu\|_2^2 \tag{4.7}$$

Because this optimal value are derived specifically under the context of DNNs training, the function $f(x)$ represents the gradient of a loss function that is used to train the parameters of a model in this particular situation.

Sampling a training data point

Suppose the training data set $D = \{x_n\}_{n=1}^N$ is a sample drawn from $p(x)$ where $p(x)$ is not known. This is the common case when we develop either a machine learning model or a DNN because we can never be able to retrieve all available data. The training data will always going to be a portion sampled from the all available data with certain probabilities. But still it is possible to define $q(x) \propto p(x)h(x)$ where $h(x) : x \to \mathbb{R}^+$ and assign a probability weight $\tilde{w}_n = h(x_n)$ to every $x_n \in D$.

To sample one training data point from $q(x)$, one firstly need to normalize the probability weights among dataset D as following:

$$w_n = \frac{\tilde{w}_n}{\sum_{n=1}^N \tilde{w}_n} \tag{4.8}$$

Then it is possible to sample from a new distribution with probabilities of $(w_1, w_2, \ldots, w_N)$ to choose the next training data point. When these probabilities equal to each other, this sampling procedure is exactly the same as previously described in naive SGD in section 4.2.1. When they are not equal, work [1] points out that the importance sampling chooses the most informative training data by optimizing the trace as following:

$$Tr(\sum(q)) = (\frac{1}{N}\sum_{n=1}^{N} w_n)(\frac{1}{N}\sum \frac{\|f(x_n)\|_2^2}{w_n}) - \|\mu\|_2^2 \tag{4.9}$$

Sampling a training batch

Sampling a batch requires the capability to evaluate $\|f(x_n)\|_2$ efficiently on each elements of the training set. Here $f(x)$ is a function from $\mathbb{R}^{d_1}$ to $\mathbb{R}^{d_2}$, and it actually computes the gradient of the loss with respect to each element. Let us replace it with $\|g(x_n)\|_2$ to make this clear. Assume one could instantly evaluate $\tilde{w}_n$ on all the training set, then it is easy to implement importance sampling in an exact manner. That is to compose the batch of size B based from training set by sampling with replacement the values of x_n with probability proportional to $\tilde{w}_n$. Let $(i_1, i_2, \ldots, i_B)$ be the indices sampled to build the batch, then the loss in one iteration using this importance sampling is shown as following:

$$L_{batch} = (\frac{1}{N}\sum_{n=1}^{N}\tilde{w}_n)\frac{1}{B}\sum_{b=1}^{B}\frac{1}{\tilde{w}_{i_m}}L(x_{i_m}) \tag{4.10}$$

The optimal trace of covariance matrix over the batch is given by:

$$Tr(\sum(q^*)) = (\frac{1}{B}\sum_{n=1}^{B}\tilde{w}_n)^2 - \|g^{TRUE}\|_2^2 \tag{4.11}$$

The constant $\|g^{TRUE}\|_2^2$ is independent on the choice of q, so it does not change as training goes. When performing batch SGD training, $\tilde{w}_n$ changes for those data points that have been trained, so this optimal value can not be used directly for sampling. However, the value given by

$$Tr(\sum(q)) = \frac{1}{B}\sum_{n=1}^{B}\|g(x_n)\|_2^2 - \|g^{TRUE}\|_2^2 \tag{4.12}$$

gives a good approximation to the $Tr(\sum(q^*))$ at each step. This indicates that using $\tilde{w}_n = \|g(x_n)\|_2$ is a good way to construct the probability array for importance sampling. The $\|g^{TRUE}\|_2$ is not a value that must be evaluated for training, but it is a good indicator that one could monitor on to see if the training is converging. When the training is getting close to a local optimal point, the value of $\|g^{TRUE}\|_2$ is getting close to zero. Sometimes the probability weights could fluctuate rapidly. Even though this fluctuation could be alleviated by normalizing over w_n, it may still be

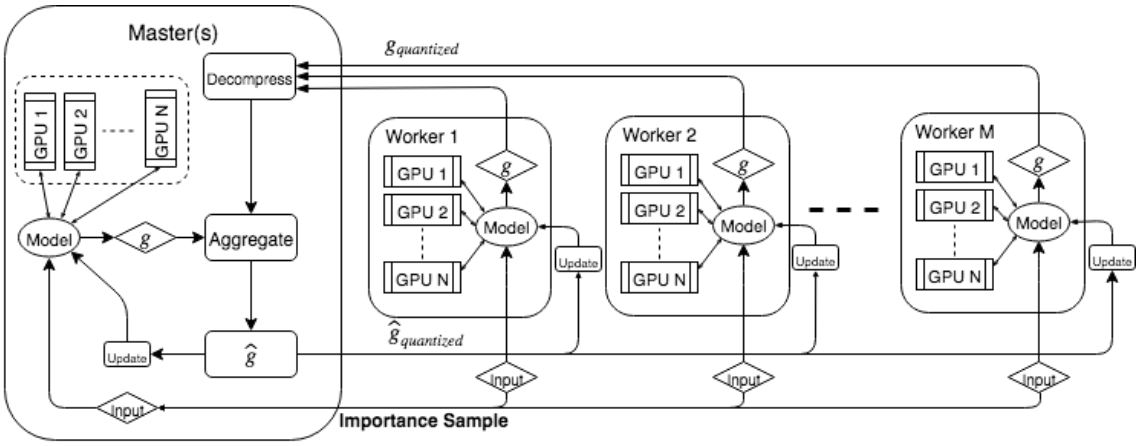


Figure 4.4: Cooperated low precision training (C-LPT) with multiple GPUs on multiple nodes.

severe when one training sample is assigned a tiny small probability weight. So to counter this effect, adding a smoothing constant before normalizing turns out to be a practical solution. The larger the constant, the more likely this will resemble a vanilla uniform sampling SGD.

4.3 Cooperated Low Precision Training (C-LPT)

C-LPT is a training paradigm designed to train compressed deep neural network using SGD algorithm. C-LPT distinguishes from traditional distributed SGD training mainly in three novel designs: gradients quantization, gradient aggregation with various precision, and training batch sampling. These three strategies are shown in Figure 4.4. We thoroughly investigate above three strategies in the remaining of this section.

4.3.1 Gradient Quantization

Quantization compress gradients computed from the deep neural network model by reducing the number of bits required to represent each gradient. Gradients are compressed on the workers side, then sent to the masters in limited bits, which afterwards will be decompressed on the masters side for aggregation. The whole procedure will be in exactly reversed order when updated gradients need to be send back to workers from masters. This compress-decompress manner is essential to reduce the whole training time as the substantial overhead of network communication is waived and the waiting time on GPU chips due to the communication bottleneck could be used for compu-

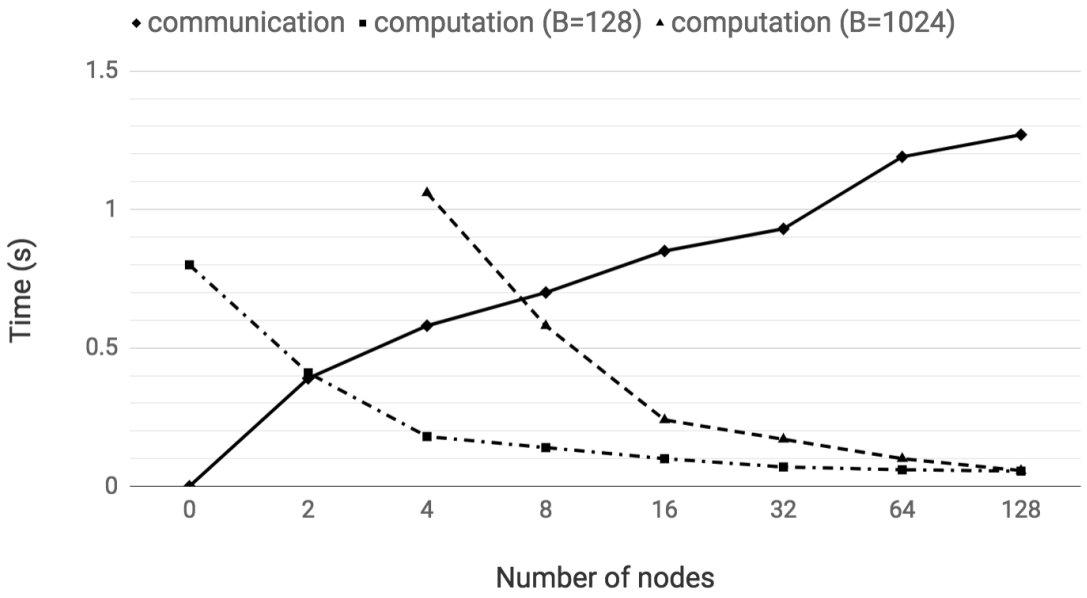


Figure 4.5: Comparison of communication time and computation time in one iteration on AlexNet when the number of nodes varies

tation. The whole training time will ideally reduce by several magnitudes along with the increases of the ratio of computation time to communication time. Figure 4.5 shows the severity of the communication bottleneck. When compute times drop below the communication times, the scalability decreases rapidly.

It has been believed that the remarkable performance of DNNs relies on the high-precision computation, and low-precision arithmetic calculation will lead to performance loss. But to what extent that this full-precision to low-precision conversion will affect the performance still remains unknown. Because this compression rate is highly important to the design of quantization algorithm, we consider two parts of a single value separately and discuss their influences on the performance.

Sign and Magnitude of Values in Back-Propagation

Back-propagation (BP) algorithm is the core of SGD. BP has two passes: forward pass to calculate gradients and backward pass to update parameters of the training model. When the same parameters must be used for two passes, BP runs in a symmetric fashion [82]. This symmetric fashion is believed to be at the foundation of BP algorithm but it is obviously against biological

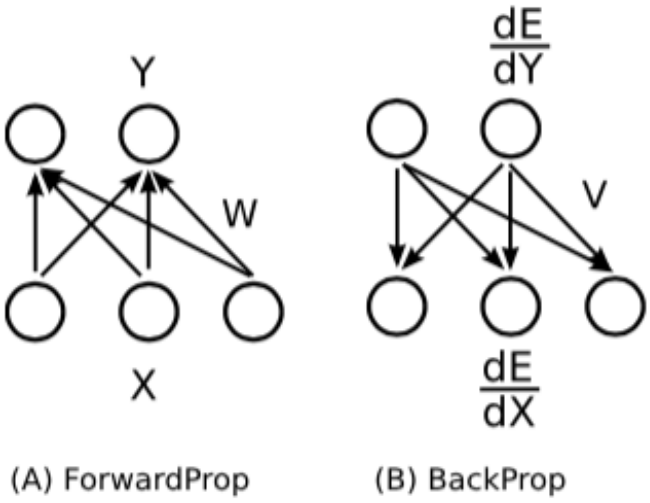


Figure 4.6: Illustration of back-propagation algorithm

plausibility. This is one of the main reasons that raises the doubt among people to the necessity of this symmetric fashion because BP works for a neural network model which is designed based on human's brain, but brain does not work in this way.

To explore whether this symmetric requirement can be relaxed, it is important to investigate whether the value can be modified without hurting the performance in term of the prediction accuracy. Each value in nowadays system is a 32-bits float. It has two components: sign which indicates whether a number is positive or negative; magnitude which is the absolute value of the number.

The BP algorithm is illustrated using a two-layer fully-connected neural network in Figure 4.6, where X and Y is the input and output, E is the objective function, W is the parameters of the model in forward pass, and V is the gradients calculated for backward pass, then we have:

$$Y_j = f_{activate}(N_j), \qquad where \;\; N_j = \sum W_{i,j} X_i \tag{4.13}$$

$$\frac{\partial E}{\partial X_i} = \sum_j V_{i,j} \frac{\partial f(N_j)}{\partial X_i} \frac{\partial E}{\partial Y_j} \tag{4.14}$$

The symmetric fashion of BP requires $V = W$. When V is modified to be other values, there can be various asymmetric BPs. There are many modification methods discussed in work [82].

Table 4.1: Comparison of the error rate (%) of SGD after sign and magnitude modification with P=0.75

dataset	SGD	uSF	brSF	frSF	brSF-p	frSF-p	RndF
MNIST	0.73	0.83	0.80	0.91	8.34	3.56	1.07
CIFAR-10	17.94	19.29	18.44	19.02	75.87	68.49	25.75
CIFAR-100	51.45	53.12	50.74	52.25	96.23	95.22	64.69

1. Uniform Sign Feedbacks (uSF): $V = sign(W)$.

2. Batchwise Random Magnitude Sign Feedbacks (brSF): $V = M \cdot sign(W)$, where M is updated for each training batch.

3. Fixexd Random Magnitude Sign Feedbacks (frSF): $V = M \cdot sign(W)$, where M is initialized once and fixed in each iteration.

4. Batchwise Random Magnitude sign-partial feedbacks (brSF-p): $V = M \cdot sign(W) \cdot P$, where sign of V is different from that of W by a probability of p.

5. Fixed Random Magnitude sign-partial feedbacks (frSF-p): $V = M \cdot sign(W) \cdot P$, where sign of V is different from that of W by a probability of p.

6. Random Feedbacks (RndF): $V \sim N(0, \sigma^2)$, where $N(0, \sigma^2)$ is a zero-mean gaussian distribution.

The first three, uSF, brSF, and frSF, keep the sign but modify the magnitude to different extent. The two after them, brSF-p and frSF-p, takes one step further, only keeps *sign* with probability P. The last one RndF replaces W totally randomly. [82] shows the results on popular datasets, MNIST [78] and CIFAR [73], in Table 4.1. The results in the table show that the magnitudes do not worsen the performance a lot. However, the performance drops a lot when signs could change with a certain probability.

This result shows that the modification on magnitudes of parameters only bring a small reduction to the performance. In comparison, maintaining the signs is more critical to achieve a high performance. Combining this conclusion with function 4.14, we can see changing the magnitude of parameters $V_{i,j}$ is equivalent as changing the magnitude of gradients $\frac{\partial E}{\partial X_i}$. This indicates, like the finding on parameters, the signs of gradients is also more critical compared to their magnitudes. On this path, algorithm Batch Manhattan (BM) is the most aggressive one. BM discards the

magnitudes of the gradients totally and only keeps the sign. This approach would not bring huge performance loss when dataset is small, but for large datasets, its performance loss is intolerable. This discovery on the importance of sign and magnitude of gradients to the performance provides an empirical support to the gradient quantization. To minimize the performance loss due to the magnitude modification, many quantization algorithms have been developed.

Binary and Ternary Quantization

Several binary and ternary quantization methods are derived directly from the idea of Batch Manhattan (BM). Some of them are used to compress the model in order to deploy neural network training on portable devices. In this case, the quantization happens on model parameters, not on gradients. On the other hand, some of them work on both parameters and gradients. No matter what, the idea behind the quantization remains unchanged.

- **BinaryConnect** [24] converts each 32-bits weight vector W_i into a binary one B_i based on the following rules:

$$B_i = \begin{cases} +1, & with\ probability\ p = \sigma(W_i) \\ -1, & with\ probability\ 1-p \end{cases} \tag{4.15}$$

  where $\sigma(x) = max(0, min(1, \frac{x+1}{2})$ is the hard sigmoid function. The full-precision weights and binarized weights both exist throughout the training. Gradients and loss are computed from binarized weights. Then the gradients will be used to update on full-precision weights. Finally function 4.15 is applied to transform full-precision weights to binary weights. This cycle repeats in next iteration.

- **Xnor-net** [94] extends the BinaryConnect and it uses a scale factor $\alpha \in \mathbb{R}^+$ to approximate the full-precision weight vector W_i. It solves an optimization problem

$$L = min\|W_i - \alpha B_i\| \tag{4.16}$$

  and gets:

$$
\begin{aligned}
B_i &= sign(W_i) \\
\alpha &= \frac{1}{d}\sum_{j=1}^{d}|W_i^j|
\end{aligned}
\tag{4.17}
$$

- **Trained Ternary Quantization** [137] introduces an extra zero as a third quantized value. It uses two symmetric thresholds $\pm\Delta_l$ and two quantization factors W_l^p and W_l^n for positive and negative weights in each layer l. Quantized ternary weights w_l^t are calculated as:

$$
w_l^t = \begin{cases} W_l^p, & w_l > \Delta_l \\ 0, & |w_l| \leq \Delta_l \\ -W_l^n, & w_l < -\Delta_l \end{cases}
\tag{4.18}
$$

One of the points that make this method different from previous work is that W_l^p and W_l^n are two independent parameters and are trained along with other parameters, that is following the rules of gradient descent as well. The derivatives are given by:

$$
\begin{aligned}
\frac{\partial L}{\partial W_l^p} &= \sum_{i\in I_i^p}\frac{\partial L}{\partial w_l^t(i)}, \quad where\ I_l^p = \{i|w_l(i) > \Delta_l\} \\
\frac{\partial L}{\partial W_l^n} &= \sum_{i\in I_i^n}\frac{\partial L}{\partial w_l^t(i)}, \quad where\ I_l^n = \{i|w_l(i) < -\Delta_l\}
\end{aligned}
\tag{4.19}
$$

Furthermore, because of the existence of two scaling factors, gradients need to be calculated with new rules as following:

$$
\frac{\partial L}{\partial w_l} = \begin{cases} W_l^p \cdot \frac{\partial L}{\partial w_l^t}, & w_l > \Delta_l \\ 1 \cdot \frac{\partial L}{\partial w_l^t}, & |w_l| \leq \Delta_l \\ -W_l^n \cdot \frac{\partial L}{\partial w_l^t}, & w_l < -\Delta_l \end{cases}
\tag{4.20}
$$

The benefits of using asymmetric trained quantization factors include firstly enabling neural networks to have higher model divergence and secondly there quantized weights acting as the multipliers to the learning rate.

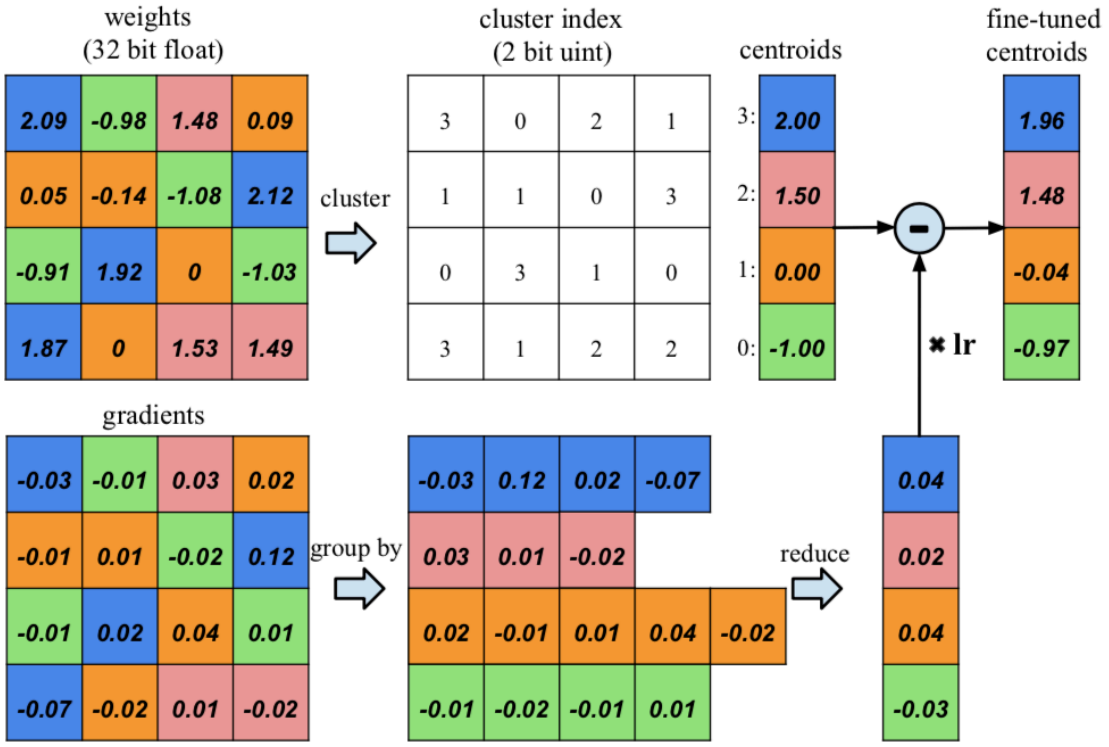


Figure 4.7: Gradients updates based on weights clustering and gradients grouping [45]

Clustering Quantization

Clustering quantization is another group of methods to quantize model parameters and gradients. It shares the similarity with previous quantization methods in keeping the sign unchanged for all values. It differs from previous methods as it chooses magnitudes for values based on clustering, which means the scale factor is not uniformly applied to values in a layer.

- **Deep Compression** [45] describes one classic clustering approach in which the gradients updates are based on weight clustering and gradients grouping. The approach is illustrates as shown in Figure 4.7.

  Imagine it is a layer with input and output both of four dimensions, so the weight matrix is 4×4. The top left is the 4×4 weight matrix, and the bottom left is the 4×4 gradient matrix. The weights are clustered into four categories, each of which is illustrated with a different color. To quantize the weights, this approach simply lets all the weights in the same cluster share the same value. During the SGD training, all the gradients are grouped by the clusters of weights according to their corresponding location and then summed up,

multiplied by the learning rate. Then this modified gradients are used to update weights to get the fine-tuned centroids on the right hand side.

- **Stochastic Quantization** [36] quantize a portion of the full-precision weights to low-bits to minimize the information loss. It firstly calculate the quantization error, and then cluster on this error to filter each element based on a quantization probability p. The final quantization result is controlled not only by this probability but also by a stochastic quantization (SQ) ratio r. The basic procedure of SQ is illustrated in Figure 4.8.

  Given the weight matrix 4×4 and a SQ ratio r, it firstly calculate the quantization error, and then derive the quantization probability p for each rows of the weight matrix. There are four different choices for choosing probability p:

  1. **Constant function**: $p_i = \frac{1}{m}$, where m is the length of probability vector. This function makes equal probability for all rows. And it ignores the quantization error randomly.
  2. **Linear function**: $p_i = \frac{f_i}{\sum_j f_j}$, where $f_i = \frac{1}{e_i + \epsilon}$. e_i is the quantization error on row i and ϵ is a very small value to avoid overflow.
  3. **Softmax function**: $p_i = \frac{exp(f_i)}{\sum_j exp(f_j)}$, where f_i is equal to the definition in linear function.
  4. **Sigmoid function**: $p_i = \frac{1}{1+exp(f_i)}$, where f_i is equal to the definition in linear function.

  Experiments show that linear function can get the smallest error rate when combined with other quantization methods. However, different functions share very close performance. This may indicate the methods in clustering error and partitioning are critical for SQ algorithm. A mixed-precision matrix is obtained after using this probability p to select a portion of quantized rows. This work performs forward and backward propagation based on this hybrid matrix during training.

Logarithmic Quantization

The low-precision numbers are meant to store using a limited number of bits that is much less than 32-bits, e.g. 4-bits or 8-bits. However, the larger the range of numbers can be represented, the better the quantization method would be. Previously described methods, such as binary and ternary

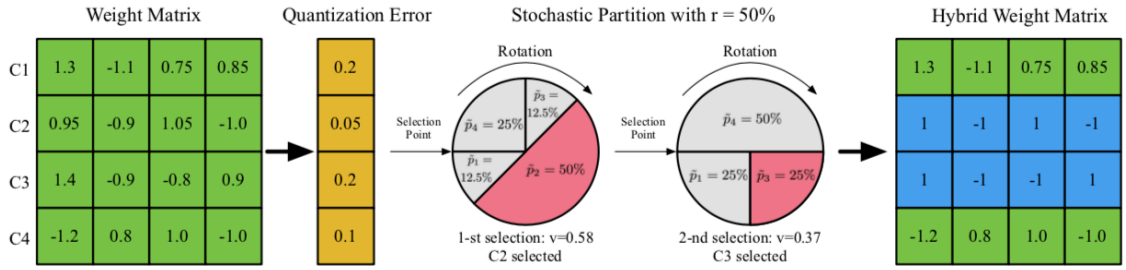


Figure 4.8: Stochastic Quantization [36]

quantization and clustering quantization, have limited capability in representing wide range of numbers. The range of numbers using binary and ternary approach is restricted by the scale factor W, while the range in clustering based methods is constrained by the number of clusters.

Ideally, logarithmic quantization techniques are developed to tackle this problem by introducing logarithm representation into the compression. It is well known that logarithmic scales reduce wide-ranging quantities to tiny scopes. When the precise value of a number is not able to be represented within 4-bits, its exponential part to the base of another fixed number may still be able to be represented using only 4-bits. For example, number '32' can not be written in 4-bits, but because it can be converted to 2^5, we can only transfer number 5 as '0101' in binary once the base, number 2, has been agreed between the sender and receiver. After that, a simple conversion could be done to restore the number '32' on receiver side.

To be compatible with the structure of computer hardware, the base of logarithmic quantization is normally chosen as two. This takes the advantage of binary bit move operations so that the high cost addition and multiplication can be replaced. The representation of a low-precision number using logarithmic quantization consists of two parts, a *scale factor* $\sigma \in \mathbb{R}$ and an *exponential factor* $b \in \mathbb{N}$. Let $D(\sigma, b)$ denote the domain of the numbers in this format of representation, then:

$$D(\sigma,b) = \{-\sigma \cdot 2^b, -\sigma \cdot 2^{b-1}, \ldots, -\sigma, -\sigma \cdot 2^{-1}, \ldots, 0, \ldots, \sigma \cdot 2^{-1}, \sigma, \ldots, \sigma \cdot 2^{b-1}, \sigma \cdot 2^b\} \tag{4.21}$$

Note that because of the characteristic of logarithm, the domain is not equally divided. Densities of values at two ends are much less than the density around zero. This uneven distribution, however, is coincidentally a perfect match with the distribution of normalized gradients. The value

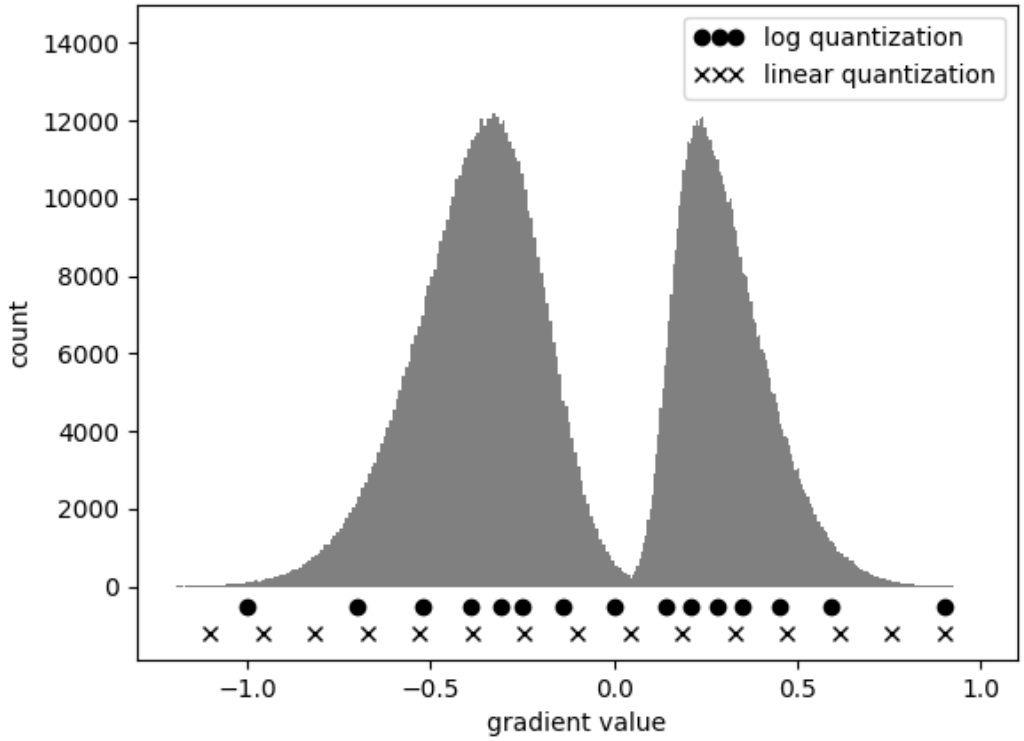


Figure 4.9: Distribution of gradients at convolutional layer after training AlexNet for 50 iterations

of gradients form a bimodal distribution (shown in Figure 4.9) with two peaks on different sides of zero. On the bottom, linear quantization and logarithmic quantization are compared, which shows that logarithmic quantization perfectly match the distribution of gradient values.

This representation is beneficial as quantized numbers with the same scale factor can be added using integer addition; two quantized numbers could be multiplied with an integer multiplication, which results in a new number whose scale factor is the product of the input scale factors and exponential factor is the sum of the input exponential factors. Because the computations on GPUs are mostly addition and multiplication, with these two efficient operations, low-precision integer arithmetic on workers side can be effectively implemented.

With the logarithmic representation being clearly shown already, we describe two methods to convert a gradient matrix G_l into a log-represented $\hat{G}_l$. This is equivalent in choosing appropriate *exponential factor* b.

- **Two thresholds**: This is to quantize each entry in the gradient matrix G into one value of following:

$$P(\sigma) = \{-\sigma \cdot 2^{n_1}, \ldots, -\sigma \cdot 2^{n_2}, 0, \sigma \cdot 2^{n_2}, \ldots, \sigma \cdot 2^{n_1}\} \tag{4.22}$$

  where $n_1, n_2 \in \mathbb{N}$ and $n_2 \leq n_1$. This idea is described in [133]. After quantization, the non-zero elements will be strictly within the range of either $[-2^{n_1}, -2^{n_2}]$ or $[2^{n_2}, 2^{n_1}]$. Suppose the target bit-width is w, then n_1, n_2 is determined as:

$$\begin{aligned} n_1 &= floor(log_2(\frac{4}{3} \times max(abs(G_l)) \\ n_2 &= n_1 + 1 - 2^{w-2} \end{aligned} \tag{4.23}$$

  where $max(abs(\cdot))$ produce the largest element with absolute value among a gradient matrix and $floor(\cdot)$ is a round down operation. For example, if $w = 3$ and $n_1 = -1$, then $n_2 = -2$. After the $P(\sigma)$ is determined, G_l is converted into a quantized one using:

$$\hat{G}_l(i,j) = \begin{cases} \beta \cdot sign(G_l(i,j)) & if \frac{\alpha+\beta}{2} \leq abs(G_l(i,j)) \leq \frac{3}{2}\beta \\ 0 & otherwise \end{cases} \tag{4.24}$$

where α, β are two adjacent element in $P(\sigma)$.

- **Rounding**: There are many rounding methods, e.g. nearest rounding and floor rounding. Following the reasons explored in works [25, 27] and the practice of work [26], *randomized rounding* seems a better choice. That is to round up or down randomly so that for any gradient g within the domain of the low-precision representation $D(\sigma, b)$, we have $\mathbb{E}[g_{quantized}] = g$. If g is out of domain $D(\sigma, b)$, $g_{quantized}$ would be the closest value to g in domain (that is the largest or smallest value).

In C-LPT, we use logarithmic quantization with randomized rounding at default, the other quantization methods are used as base line for evaluation in empirical studies. The bit-width w is chosen between 4, 8, and 16 bits to distinguish from full-precision 32-bits. This bit-width w only restricted to the *exponential factor* b as *scale factor* σ only will be updated by *bit centering* strategy (described in section) every several iterations. This scale factor σ do not need to be transferred every time. Also because it is a universal value among all gradient layers, only one value will be send. This should not be taken into consideration of communication bottleneck. Suppose w is chosen to be 4, then 4-bits quantization is performed in following manner. One bit will be used to represent *sign*, and the remaining 3 bits to represent at most 8 different values for the power of two, that is the smallest gradient could be $\pm 2^{-7}$. In this way, the *exponential factor* is in a linear scale. An even more aggressive quantization would be transform this exponential factor into log scale as well such that a two-level logarithmic quantization is performed. Although this would be more effective, the difficulty in implementation may cause extra computation overhead in the end which counteracts the benefits of communication reduction.

4.3.2 Gradient Aggregation with Various Precision

Gradient aggregation is a very important step in C-LPT paradigm because a well-designed aggregation algorithm is critical to achieve the desired linear rate of convergence of SGD without increasing per-iteration training costs. In C-LPT, we introduce a sub-sampling strategy to choose a portion of training batch on workers side to be trained on masters side. Because the gradients calculated by BP algorithm on masters side do not need to be sent over network, these gradients

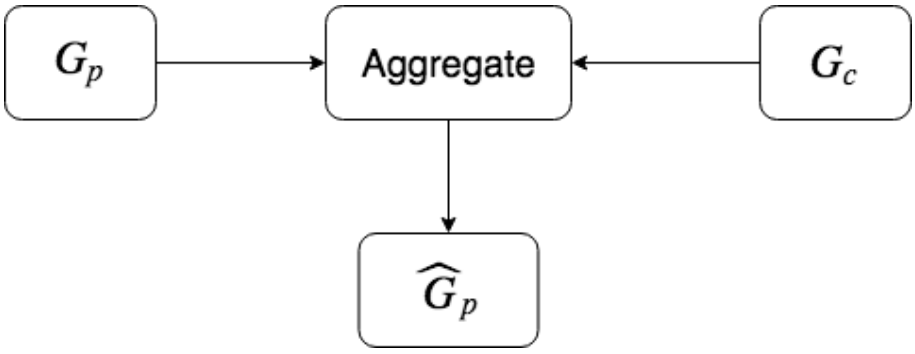


Figure 4.10: Gradient aggregation with different precision gradient matrix on master side

can safely keep the full-precision representation. On the other hand, the gradients calculated on workers side do need to be sent to masters via bandwidth limited network, these parts are quantized using logarithmic quantization before transporting. Even though these quantized gradients will be de-quantized on masters, they are still discrete values. The illustration of this aggregation is shown in Figure 4.10. This results in aggregation with various types of values: continuous full-precision values denoted as G_p, and discrete low-precision values denoted as G_c. The result of aggregation will be continuous full-precision values denoted as $\hat{G}_p$, and these values will be quantized again to send back to workers in order to update models on workers.

In quantized vanilla SGD (described in section 4.2.1, Figure 4.3), only the quantized gradients received from workers are aggregated. This is essentially different from the scenario in C-LPT as only one type of values are summed up and each gradient are treated equally due to the uniform batch sampling. So the average aggregation strategy is intuitively the best one. However, in C-LPT, we adopt non-uniform sub-sampling strategy to choose data points. This gives us more choices in trying different aggregation approaches other than averaging. We discuss three categories of aggregation methods in this section. In empirical studies, averaging is used as baseline aggregation algorithm.

We propose an importance-awareness gradient aggregation (IAGA) algorithm that incorporates the idea of *reusing* borrowed from two classic variance reduction algorithms and *bit-centering* to be independent of quantization bit-width. Before we show the details of IAGA, we briefly discuss coefficient combination strategies and these two classic variance reduction algorithms.

Linear Combination

Intuitively we can perform a linear combination of G_p and G_c in order to get $\hat{G}_p$. This can be written as:

$$\hat{G}_p = \alpha \cdot G_p + \beta \cdot G_c, \quad where \ \ \alpha, \beta \in [0, 1] \tag{4.25}$$

α, β can be seen as two aggregation weights that are assigned to G_p, G_c, indicating how much importance being put on gradients calculated on masters side and workers side. They can be set as fixed values over all iterations or dynamic values that is related to iteration number i.

- **Fixed Values**: We discuss three special sets of values:

  1. Set $\alpha, \beta = 0.5$. This is equivalent to compute average between G_p and G_c. This would result in a slightly better performance than just averaging among G_c because G_p provides a high precision of gradient for part of data points. This makes up the precision loss due to quantization.
  2. Set $\alpha = 0, \beta = 1$. This reduce the problem to quantized vanilla SGD. The gradients calculated on the masters side do not contribute to the training at all.
  3. Set $\alpha = 1, \beta = 0$. This is the opposite of above scenario. It wastes the computation power on workers side totally. By setting values in this way, the model is trained on full-precision, only with a relatively small amount of data points in each iteration. This actually leads to a better model in terms of prediction accuracy. But the disadvantage is obvious that this would take way longer time in training as more iterations are needed.

  Three sets of values take different trade-offs between efficiency and effectiveness. This shows a big picture of the relationships between α, β and the performance. Basically, fixed values are not as good as dynamic values setting.

- **Dynamic Values**: Dynamic changing the relative value of α, β based on iteration i normally outperforms fixed values. The idea is to design a function $f(\cdot)$ on i to produce a weight for G_p. And $1 - f(i)$ for G_c, that is:

$$\hat{G}_p = f(i) \cdot G_p + (1 - f(i)) \cdot G_c, \quad where \ \ f(i) \in [0, 1] \tag{4.26}$$

1. Linear change rate $f(i) = \frac{i}{T}$, where T is the total iteration number. This is to set the weight of G_p to increase linearly from 0 to 1 throught out the training.
2. Exponential increasing rate $f(i) = \frac{exp(i)}{exp(T)}$, where T is the total iteration number. This is to set a slow increase rate to the beginning of the training, but the G_p dominated the gradient update when training is close to the end.
3. Sigmoid change rate $f(i) = \frac{1}{1+exp(T/2-i)}$, where T is the total iteration number. This is to set a rapid increase rate to G_p in the mid-phase of training which is the most difficult part at most of the time. The increase rate in the beginning and ending phase of the training is relatively low as in the fine-tuning phase, the gradient values better not experience sharp changes.

We know that thorough out the training procedure, the learning is quick at the beginning and slows down as iteration numbers increase. With three change rates handy, we could switch between them based on the learning curve in order to fine tune the model to achieve better performance.

Variance Reduction Aggregation

As discussed earlier, the inherent variance impedes the convergence of SGD. Computing more stochastic gradient through more data points, e.g. increasing batch size, is a possible way to reduce variance, because when the batch size is as large as training dataset, it reduces to gradient descent (GD) training. The idea of sub-sampling a portion to be trained on masters side is motivated by this method. Another possible way to reduce variance is by *reusing* previously computed information and adding this information into the aggregation stage [12]. The reasoning is that if the current iteration is not too far from previous iterations, then the gradients computed in previous stochastic iterations may still be of guidance for the update at this iteration. To be more specific, if gradients information are indexed and kept in storage, then the aggregation result $\hat{G}_p$ could be revised further.

- **Stochastic Variance Reduction Gradient (SVRG)** [62] SVRG operates in cycles where each cycle contains several iterations. When a cycle starts, the algorithm computes a batch gradient:

$$\nabla F_n(w_k) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(w_k) \tag{4.27}$$

Then initializing $\tilde{w}_1 = w_k$. After this, m inner iterations run to update w_j as:

$$\begin{aligned} \tilde{w}_{j+1} &= \tilde{w}_j - \mu \tilde{g}_j \\ \tilde{g}_j &= \nabla f_{i_j}(\tilde{w}_j) - (\nabla f_{i_j}(w_k) - \nabla F_n(w_k)) \end{aligned} \tag{4.28}$$

where $i_j \in \{1, \ldots, n\}$ is chosen at random. The reason why this works can be explained in a simple way. Because $\mathbb{E}[\nabla f_{i_j}(w_k)] = \nabla F_n(w_k)$ over all $i_j \in \{1, \ldots, n\}$, one can see $\nabla f_{i_j}(w_k) - \nabla F_n(w_k)$ as the bias in the gradient estimate $\nabla f_{i_j}(w_k)$. Therefore, the algorithm chooses a gradient $\nabla f_{i_j}(\tilde{w}_j)$ at random to be evaluated at inner iterate j and revise it based on a certain bias. So, $\tilde{g}_j$ acts as an unbiased estimator of $\nabla F_n(\tilde{w}_j)$ but with a variance expected to be much smaller than simply choosing $\tilde{g}_j = \nabla f_{i_j}(\tilde{w}_j)$, which is the default value in vanilla SGD. Given the algorithm at cycle k, we have:

$$\begin{aligned} \mathbb{E}[F_n(w_{k+1}) - F_n(w_*)] &\leq \rho \cdot \mathbb{E}[F_n(w_k) - F_n(w_*)] \\ \rho &= \frac{1}{1 - 2\mu L}\left(\frac{1}{mc\mu} + 2\mu L\right) < 1 \end{aligned} \tag{4.29}$$

where μ is learning rate, c is the convex coefficient, and L is Lipschitz continuous coefficient. SVRG is shown in Algorithm 3. This indicates that there is a linear convergence rate for the outer iterates w_k, and from w_k to w_{k+1}, it requires $2m + n$ evaluations of gradients.

In practice, SVRG is proved to be quite effective in certain applications where SGD runs for a good number of epochs. The length of inner cycle m and learning rate μ is determined experimentally without knowing exact c and L.

- **Stochastic Average Gradient Aggregation (SAGA)** [30] SAGA does not operate in cycles and does not compute batch gradients either. Instead, it computes a stochastic vector g_k as the average of stochastic gradients evaluated at previous iterates at each iteration as described in [12]. For example, in iteration k, SAGA has $\nabla f_i(w_{[}i])$ for each $i \in \{1, \ldots, n\}$ stored. Here $w_{[}i]$ is the value calculated by ∇f_i in the latest iteration. Another random variable $j \in \{1, \ldots, n\}$ is chosen and the g_k is updates as:

Algorithm 3 SVRG: Stochastic Variance Reduction Gradient

Input: N loss gradients ∇f_i, number of cycles K, cycle length M, and learning rate μ
Output: $\tilde{w}_k$
  for $k = 1$ to K **do**
    $\tilde{g}_k = \nabla f(\tilde{w}_k) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\tilde{w}_k)$
    $w_{k,0} = \tilde{w}_k$
    for $t = 1$ to M **do**
      sample i uniformly from $\{1, \ldots, n\}$
      $w_{k,t} = w_{k,t-1} - \mu(\nabla f_i(w_{k,t-1}) - \nabla f_i(\tilde{w}_k) + \tilde{g}_k)$
    end for
    $\tilde{w}_{k+1} = w_{k,M}$
  end for
  return $\tilde{w}_{K+1}$

$$g_k = \nabla f_j(w_k) - \nabla f_j(w_{[j]}) + \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(w_{[i]}) \tag{4.30}$$

Taking $\mathbb{E}[g_k]$ with respect to all $j \in \{1, \ldots, n\}$, we will have $\mathbb{E}[g_k] = \nabla F_n(w_k)$. Therefore, the method has unbiased gradient estimates, and the variance is smaller than vanilla SGD. More specifically, the convergence rate is given by:

$$\begin{aligned} \mathbb{E}[\|w_{k+1} - w_*\|_2^2] &\leq (1 - \frac{c}{2(cn+L)})^k \cdot (\|w_1 - w_*\|_2^2 + \frac{nD}{cn+L}) \\ D &= F_n(w_1) - \nabla F_n(w_*)^T(w_1 - w_*) - F_n(w_*) \end{aligned} \tag{4.31}$$

The most interesting part about SAGA is that it enjoys a linear rate of convergence by using randomized *reusing* of previous gradient information.

Importance Awareness Gradient Aggregation (IAGA)

We describe our proposed algorithm IAGA in this section. Using the linear combination aggregation rule shown in equation 4.25, our IAGA algorithm runs in two separate steps. On one hand, for the first step, we design a sophisticated method in choosing coefficient α and β so that these two factors are aware of the importance of G_p and G_c in a specific iteration. That is why we call it *importance awareness gradient aggregation*. On the other hand, for the second step, we need to consider how to aggregate gradients from different workers and batches on masters to get G_c and

G_p respectively.

- **Choosing Coefficient**

  Factors α, β acts as weights in aggregating G_p and G_c. Before we describe the algorithm in choosing α and β. Let us take a closer look at the characteristic of G_p and G_c.

  G_p is calculated from a relatively small portion of training batch in each iteration. Because of this, it lacks of exploration in different gradient descent directions. In another word, the gradient descent direction given by G_p may not be the best one. However, because these part of gradients do not need to be sent through network, they enjoys high precision. That is to say, it has the best estimation of the current model's distance to a local optimum.

  In comparison, G_c is calculated from the whole training batch. So it has already explored more gradient descent directions than G_p. It may have produced a faster way to get to the local optimum than G_p. But, because these gradients are sent to the masters in quantized value, they may not be accurate in the magnitude along this path. Still it could reflect the best estimation to the gradient variance of the current iteration.

  We use E to denote the estimate of distance from current model to a local optimum, and we use F to denote the estimate of gradient variance.

  To esimate E, in iteration t, we calculate $\overline{g}$ and $\overline{g^2}$ by keeping track of g_t and $g_t \bigodot g_t$. These are the first and second order of gradient. We also have $Var(g_t) = \mathbb{E}g_t^2 - (\mathbb{E}g_t)^2$ as described in work [130].

  To estimate F, because of the fact that $\nabla f(x) \leq \|H\|\|x - x^*\|$ for a quadratic $f(x)$ with Hessian matrix H and optimum x^*, we calculate $\overline{h}$ and $\overline{\|g\|}$ as the average of curvature h_t and gradient norm $\|g_t\|$. Then we have:

$$E = \frac{\overline{\|g\|}}{\overline{h}}, \quad F = \|\overline{g^2} - \overline{g}^2\|_1, \quad where \ \overline{h} = avg(h_t), \overline{\|g\|} = avg(\|g_t\|) \tag{4.32}$$

  To get exact α, β in iteration t. We formulate it as an optimization problem:

$$\begin{aligned} \alpha_t, \beta_t &= \arg\min_{\alpha,\beta} \alpha_t \frac{E_{p,t}}{E_{c,t}} + \beta_t \frac{F_{p,t}}{F_{c,t}} \\ s.t. \quad \alpha &\geq (\frac{\sqrt{h_{max}/h_{min}} - 1}{\sqrt{h_{max}/h_{min}} + 1})^2 \\ \beta &= \frac{(1-\sqrt{\mu})^2}{h_{min}} \end{aligned} \tag{4.33}$$

where $h_{max}, h_{min} = max(h_t), min(h_t)$, respectively, and $h_t = \|g_t\|^2$. h_t is an eigenvalue with eigenvector g_t and it is also an estimation to the curvature of Hessian matrix along gradient direction g_t. Recall in section 4.2.2, $g(x_n)$ is a good estimator to the importance of a training data point x_n. Here g_t is a part of E and F, used to estimate the importance of G_p and G_c. Algorithm 4 shows how to calculate α, β.

Algorithm 4 Coefficient Computing in IAGA

Input: $\alpha_0 = 0, \beta_0 = 1, h_{max} = h_{min} = 0, \lambda = 0.1$
Output: α_t, β_t
  for $t = 1$ to T **do**
    $h_t = \|g_t\|^2$
    $h_{max} = \lambda \cdot h_{max} + (1-\lambda) \cdot max(h_t)$
    $h_{min} = \lambda \cdot h_{min} + (1-\lambda) \cdot min(h_t)$
    $E_p, E_c = \text{DistanceEstimate}(G_p, G_c)$ ▷ Evaluating using equation 4.32
    $F_p, F_c = \text{VarianceEstimate}(G_p, G_c)$
    $\alpha, \beta = \arg\min_{\alpha,\beta} \alpha_{t-1} \cdot E_{p,t}/E_{c,t} + \beta_{t-1} \cdot F_{p,t}/F_{c,t}$
    $\alpha_t = \lambda \cdot \alpha_{t-1} + (1-\lambda) \cdot \alpha$ ▷ Accumulate coefficient to simulate momentum fashion
    $\beta_t = \lambda \cdot \beta_{t-1} + (1-\lambda) \cdot \beta$
  end for
  return α_t, β_t

- **Computing G_p and G_c**

  We discuss how to compute G_c by combining the low-precision representation with SVRG in detail. G_p is easier compared with G_c because there is no quantization involved. Previous work [26] propose Low-precision SVRG (LP-SVRG) as a fully quantized version, where the whole model is quantized as well. Because in our algorithm, we basically do not quantize the model, only focusing on quantizing the gradients would be enough. So we could borrow the basic idea from it but modify it to suit our scenario. Gradient-only low-precision SVRG (GL-SVRG) should store the model vector as full precision, but the gradients that need to be send to masters are in low-precision representation (σ, b). GL-SVRG is shown in

Algorithm 5.

By quantizing the gradients, GL-SVRG will not converge asymmtotically at a linear rate. This is because when gradients get closer to the limitation of representation, it can not get any closer, and the convergence stops. We assume that GL-SVRG will converge linearly before it gets to this limit. This requires that the objection function f to be c-strongly convex, that is:

$$(x-y)^T(\nabla f(x)-\nabla f(y)) \geq c \cdot \|x-y\|^2 \tag{4.34}$$

and the gradients ∇f_i of f_i should be all L-Lipschitz continuous:

$$\|\nabla f_i(x)-\nabla f_i(y)\| \leq L\|x-y\| \tag{4.35}$$

the condition number is defined as $k = L/c$. To derive the theoretical result, we modify the **update** rule in Algorithm 5 to:

$$\tilde{w}_{k+1} = w_{k,m}, \quad where \;\; m \;\; \text{uniformly sampled from} \{0,\ldots,M-1\} \tag{4.36}$$

Theorem 4.1. *Suppose we set the learning rate μ and cycle length M to be:*

$$\mu = \frac{\gamma}{4L(1+\gamma)} \qquad M \geq \frac{8k(1+\gamma)}{\gamma^2} \tag{4.37}$$

where $\gamma \in (0,1)$, then GL-SVRG converges to the limit at a linear rate of:

$$\mathbb{E}[f(\tilde{w}_{K+1})-f(w^*)] \leq \gamma^K(f(\tilde{w}_1)-f(w^*)) + \frac{2d\sigma^2 L}{\gamma(1-\gamma)}, \quad where \;\; w \in \mathbb{R}^d \tag{4.38}$$

So after running K cycles, we will converge at a linear rate to a distance ϵ from the limit, where $k = log(\frac{f(\tilde{w}_1)-f(w^*)}{\epsilon})$. We can see from equation 4.38, as the number of bits b becomes smaller, a larger σ should be used to satisfy equation 4.38. This will make the limit even worse.

Algorithm 5 GL-SVRG: Gradient-Only Low-Precision SVRG

Input: N loss gradients ∇f_i, number of cycles K, cycle length M, and learning rate μ
Output: $\tilde{w}_k$

```
for k = 1 to K do
    ∇f(w̃_k) = (1/n) Σ_{i=1}^{n} ∇f_i(w̃_k)
    quantize g̃_k = Q_(σ,b)(∇f(w̃_k))          ▷ Assume Q_(σ,b) is the quantization function
    w_{k,0} = w̃_k
    for t = 1 to M do
        worker side:
            sample i uniformly from {1, ..., n}
            quantize g_{k,t-1} = Q_(σ,b)(∇f(w̃_{k,t-1}))
        master side:
            g_{k,t} = g_{k,t-1} − ∇f_i(w̃_k) + g̃_k
            quantize g̃_{k,t} = Q_(σ,b)(g_{k,t})
        worker side:
            w_{k,t} = w_{k,t-1} − μ g̃_{k,t}
    end for
    update w̃_{k+1} = w_{k,M}
end for
return w̃_{K+1}
```

Bit centering To get over the limitation hurdle described before in order to achieve a better accuracy level, and to converge when using any arbitrary precision quantization, we borrow the *bit centering* technique from work [26]. Bit centering will reduce the noise from quantization as the training converges. Combining the update rules in both outer iteration and inner iteration, the gradient update in SVRG can be re-written as:

$$\frac{1}{n}\sum_{i=1}^{n}(f_i(w) - (w-\tilde{w})^T\nabla f_i(\tilde{w}) + (w-\tilde{w})^T\nabla f(\tilde{w})) \tag{4.39}$$

If we substitute w with $w = \tilde{w} + z$, then turn the objective to be a minimized optimization over z, then we have:

$$f(\tilde{w}+z) = \frac{1}{n}\sum_{i=1}^{n}(f_i(\tilde{w}+z) - z^T\nabla f(\tilde{w})) \tag{4.40}$$

When $\tilde{w}$ is getting closer to the optimum $\tilde{w}$, variable z is still far from optimized. Therefore, we still need to optimize z more carefully within a smaller area. Assume the objective function is c-strongly convex, we have:

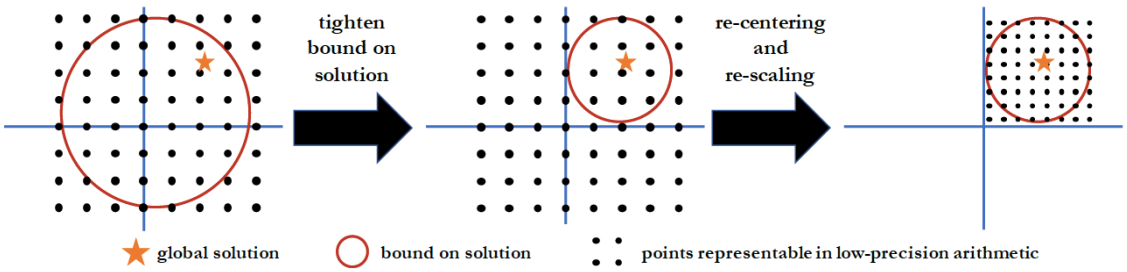


Figure 4.11: Illustration of bit centering [26] technique

$$\|z^*\| = \|\tilde{w} - w^*\| \leq \frac{1}{c}\|\nabla f(\tilde{w})\| \tag{4.41}$$

This tells us that $z^* = w^* - \tilde{w}$ will be guaranteed to be within the domain of (σ, b) if we dynamically reset the quantization domain from (σ, b) to:

$$(\sigma^{'}, b) = (\frac{\|\nabla f(\tilde{w})\|}{c(2^{b-1}-1)}, b) \tag{4.42}$$

This can be explained as following: When the magnitude of $\nabla f(\tilde{w})$ becomes smaller, it can still be represented with lower-magnitude error even though the number of bits used does not change. The representation approach is to make σ smaller along with the $\nabla f(\tilde{w})$. This procedure can be illustrated using Figure 4.11. As the algorithm converges, we bound the solution within a smaller and smaller ball. Periodically, the points are re-centered on this ball, and then re-scaled so that more points are inside the ball [26]. This decreases the quantization error as training converges.

Adding *bit centering* technique to GL-SVRG, we show new algorithm GL-SVRG-BC in Algorithm 6. Most of the algorithm structure is remained same, except that we use $z = w - w^*$ to replace w so that z is stored in low-precision. Through this modification, we are able to show that GL-SVRG-BC converge as at linear rate and it is able to get over the quantization error limitation.

Theorem 4.2. *Suppose we set the learning rate μ and cycle length M to be:*

$$\mu = \frac{\gamma}{4L(1+\gamma)} \qquad M \geq \frac{8\kappa(1+\gamma)}{\gamma^2 - 2\kappa^2 d(1+\gamma)(2^{b-1}-1)^{-2}} \tag{4.43}$$

Algorithm 6 GL-SVRG-BC: Gradient-Only Low-Precision SVRG with Bit Centering

Input: N loss gradients ∇f_i, number of cycles K, cycle length M, and learning rate μ
Output: $\tilde{w}_k$

```
for k = 1 to K do
    ∇f(w̃_k) = (1/n) Σ_{i=1}^{n} ∇f_i(w̃_k)
    σ_k = ‖∇f(w̃_k))‖ / (c(2^{b-1} − 1))
    re-scale (σ, b) = (σ_k, b)
    quantize g̃_k = Q_{(σ,b)}(∇f(w̃_k))          ▷ Assume Q_{(σ,b)} is the quantization function
    z_{k,0} = z̃_k
    for t = 1 to M do
        worker side:
            sample i uniformly from {1, ..., n}
            quantize g_{k,t-1} = Q_{(σ,b)}(∇f(w̃_{k,t-1} + z_{k,t-1}))
        master side:
            g_{k,t} = g_{k,t-1} − ∇f_i(w̃_k) + g̃_k
            quantize g̃_{k,t} = Q_{(σ,b)}(g_{k,t})
        worker side:
            z_{k,t} = z_{k,t-1} − μ g̃_{k,t}
    end for
    update w̃_{k+1} = w̃_k + z_{k,M}
end for
return w̃_{K+1}
```

where $\gamma \in (0,1)$*, and the number of bits used satisfy:*

$$b \geq 1 + log_2(1 + \sqrt{\frac{2\kappa^2 d(1+\gamma)}{\gamma^2}}) \tag{4.44}$$

then GL-SVRG-BC converges to the limit at a linear rate of:

$$\mathbb{E}[f(\tilde{w}_{K+1}) - f(w^*)] \leq \gamma^K (f(\tilde{w}_1) - f(w^*)) \tag{4.45}$$

This theorem demonstrates that the algorithm enjoys a linear convergence rate with constant bit-width low precision representation within the inner cycle. As it is discussed in [26], this also suggests that low-precision training should be combined with techniques to improve the condition number because as the condition number k gets larger, longer epoch length M is needed.

In gradient aggregation part of C-LPT paradigm, we developed IAGA algorithm which computes the dynamic aggregation coefficients based on estimation of the distance of current model to the local optimum and the variance of gradient. IAGA evaluates the G_p and G_c using modified SVRG algorithm to be adaptive to various bit-widths. It enjoys a linear convergence rate when the objective function is strongly convex. This is better than the sublinear convergence rate of vanilla SGD. We demonstrate the performance of IAGA in empirical studies.

4.3.3 Training Batch Sampling

In each iteration of stochastic gradient descent (SGD), it typically requires a batch uniformly sampled from the whole dataset, then distribute this batch to different workers. This is not easy for implementation on system level. ImageNet dataset [74] has more than one million full resolution RGB images of size 256×256. This takes roughly 256 GB on the disk. Disk I/O is the most remarkable bottleneck when uniformly sampling a batch from this dataset. Furthermore, large volume of disk read will slow down the training as the processors are idle during the reading.

As a result, in practice, the random sampling is not directly drew from disk, but performed in a pre-permuting fashion. For training on large datasets, there is often a data server involved in charge of organizing the training batch on each worker in each iteration. Before the each epoch

starts, the master pre-permutes the entire dataset $D = \{d_1, \ldots, d_N\}$ of size N by indices of images $\{1, 2, \ldots, N\}$, then slices permutation into several batches.

$$Permute(D) \rightarrow D = \{B_1, B_2, \ldots, B_T\} \tag{4.46}$$

where $B_i = \{d_{i_1}, \ldots, d_{i_b}\}$ is a batch of data points trained in one iteration. T and b is iteration number and batch size, respectively. In iteration t, the batch B_t is further sliced into several parts to be trained on each worker. Worker w will get a set of training data points as $\{d^w_{t_1}, \ldots d^w_{t_m}\}$. During the training, data server send these pre-permuted data points to workers in each iteration in a sequential way; restart the permutation when one epoch ends. Because this results in a sequential reads on disk, it significantly reduces the disk I/O time compared to previous random reads.

Alternatively, existing deep learning frameworks such as Caffe [61] only permute once before training and restart fetching a batch when one epoch ends. The batch assigned to iteration t is B_t, where t is calculated as

$$t = (T * epoch + t) mod \frac{N}{b} \tag{4.47}$$

This creates a fixed pseudo random sampling and it treats each batch equally. The problem of this sampling pattern lies in the consistent gradient updates on batches without considering the training stages. Each batch has a fixed gradient update, but updates between batches are largely different. Not only different batches, but also different data points correspond to different loss. Their gradient magnitudes and directions can be significantly different. This is due to the truth that some data points are more different to classify. It is useless to update a batch with small loss as frequently as a batch with large loss. This motivates the use of importance sampling to sample the most informative data points at each iteration.

In C-LPT, we combine the uniform sampling and importance sampling together to accelerate the training. The importance sampling happens only when sampling a subset of the training batch on workers to be trained on the master side. This particular batch on workers side is still uniformly sampled from the whole dataset. We use Figure 4.12 to illustrate this procedure. In iteration t, suppose our scheduled training batch is $B_t = \{d_1, \ldots, d_{10}, d_{30}, \ldots, d_{40}\}$. By slicing this batch, each worker gets their own training portion as $w_1 = \{d_1, \ldots, d_{10}\}$ and $w_2 = \{d_{30}, \ldots, d_{40}\}$ for

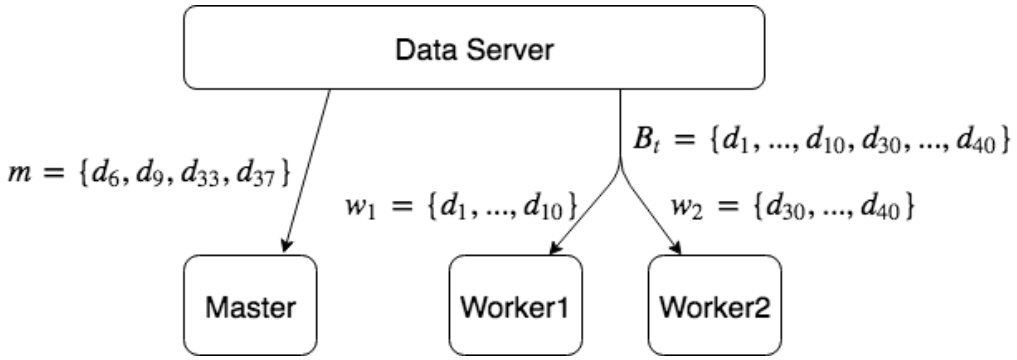


Figure 4.12: Illustration of importance sampling

$worker1$ and $worker2$ respectively. These three training batches B_t and w_1, w_2 are all uniformly sampled from the entire dataset. On master side, we assume data points $\{d_6, d_9, d_{33}, d_{37}\}$ have higher importance than other data points. Therefore, data server send this portion of data to be trained on master side. Now, we describe the method we used to compute the importance.

In section 4.2.2, we pointed out that it is possible to gain a speedup by sampling from a distribution that minimizes the trace $Tr(\sum(q))$ and it has been shown that the optimal distribution is proportional to the gradient norm $\|g(x_n)\|$ of each data point, where x_n is one data point. However, computing such distribution while training is prohibitively expensive in term of both computational cost and storage space.

In order to compute the importance for the whole dataset, we use the technique proposed in [67], where two steps of pre-samples are used to update the importance distribution only when the variance reduction is possible. It pre-samples a large batch of data points and computes the importance distribution on that batch, then re-sample a smaller batch with replacement. When the size of large batch is B and the size of smaller batch is b, it reports a maximum variance reduction of $\frac{1}{b^2} - \frac{1}{B^2}$. This variance reduction is proportional to the L_2 norm between importance distribution g and the uniform distribution u. The squared L_2 distance can be shown as

$$\|Var_g - Var_u\|_2^2 = (\frac{1}{B}\sum_{i=1}^{B}\|g(x_i)\|_2)^2 B\|g-u\|_2^2 \tag{4.48}$$

To be efficient in choosing a threshold on this squared L_2 distance, we can design a hyperparameter τ by dividing the variance reduction with the original variance. That is to let

$$
\begin{aligned}
1 - \frac{1}{\tau^2} &= \frac{\|Var_g - Var_u\|_2^2}{\frac{1}{B}\sum_{i=1}^{B}\|g(x_i)\|_2^2} \\
&= \frac{(\frac{1}{B}\sum_{i=1}^{B}\|g(x_i)\|_2)^2 B\|g-u\|_2^2}{\frac{1}{B}\sum_{i=1}^{B}\|g(x_i)\|_2^2} \\
&= \frac{1}{\sum_{i=1}^{B} g(x_i)^2}\|g-u\|_2^2 \\
\Longleftrightarrow \frac{1}{\tau} &= \sqrt{1 - \frac{1}{\sum_{i=1}^{B} g(x_i)^2}\|g-u\|_2^2}
\end{aligned}
\tag{4.49}
$$

We can set a threshold τ_{th} to decide when to update the importance distribution. Intuitively, τ_{th} can be choosed as $\frac{B+3b}{3b}$ if assuming the back propagation requires twice the amount of the time as the inference. When $\tau \geq \tau_{th}$, the importance distribution is updated iteratively until converge. Algorithm 7 shows the procedure of importance computing.

Algorithm 7 Compute Importance Distribution

Input: batch size on workers side B, training batch size on master b, threshold τ_{th}, smooth average parameter a, initial model parameter W_0
Output: trained model W

```
while W is not converged do
    if τ ≥ τ_th then                                  ▷ Update importance distribution
        U ← B uniformly sampled data points
        compute importance ‖g(x_i)‖      ∀i ∈ U
        G ← b sampled according to importance of ‖g(x_i)‖ from U
    else
        G ← b importance sampled from U
    end if
    master: W_t^m ← sgd_step(G, W_{t-1})
    workers: W_t^w ← sgd_step(U, W_{t-1})
    gradient aggregation: W_t ← aggregate(W_t^w, W_t^m)
    update τ: τ ← aτ + (1 − a)(1 − 1/(Σ_{i=1}^B g(x_i)^2) ‖g − 1/|U|‖_2^2)^{-1/2}
end while
return W_T
```

4.4 Experiments

In this section, we empirically evaluate the performance of C-LPT paradigm. C-LPT is different from other distributed training methods in three core designs. For the gradient quantization part,

our goal is to validate that the communication overhead is largely reduced by adopting quantization techniques, and therefore the increased throughput can lead to a faster training procedure compared to full precision training. For the various precision aggregation part and training batch sampling part, our goal is to show that these algorithms using low-precision representation can still produce a competitive high accuracy as the traditional SGD method and the optimized convergence rate can accelerate the end-to-end training time as well.

4.4.1 Experimental Settings

We validate the performance of C-LPT with two types of machine learning tasks, i.e., image classification and speech recognition. We run image classification with convolutional neural network (CNN) architectures on three datasets MNIST [78], Cifar-10 [73], and ImageNet [74]. And we run speech recognition with recurrent neural network (RNN) architecture on Librispeech ASR corpus [90]. The network architectures we used in this empirical studies have been described in Section 2.2.2. Now we briefly describe the datasets.

MNIST. This dataset contains handwritten digits only. There are 60000 training images and 10000 test images. Each image is of size 28×28 and is in gray scale. The whole dataset is divided into ten classes representing ten digits from 0 to 9. Compared to other two image datasets, this one is the easiest one because it is relatively small. We use MNIST to initially validate the practicality of our ideas, then used the other two datasets to further demonstrate the performance of each idea.

Cifar-10. Images in this dataset are color images in RGB mode rather than in gray scale. There are 50000 training images and 10000 test images. Each image is of size 32×32. It normally takes several hours to train a CNN model before convergence.

ImageNet. This is the largest dataset among image classification tasks. The training dataset contains over one million images labeled in 1000 classes. The validation dataset contains 50000 images, equally distributed in each class.

Librispeech ASR corpus. This dataset contains large scale corpus of read English speech. The length of speech is about 1000 hours. We use DeepSpeech 2 architecture without n-gram language model. It has several converlutional layers followed by a 7-layer gated recurrent unit (GRU) of 1200 hidden units per layer to test the word error rate.

We run experiments using TensorFlow framework. We train our models with popular SGD

Table 4.2: Volume of message sent over network

Network	Algorithm	Bit-width	Message Size(MB)	Reduction Ratio
AlexNet	No Compression	32	232.56	1 ×
	Binary	2	23.83	10 ×
	Ternary	3	26.18	9.1 ×
	Logarithm	5	**34.90**	**8.2** ×
VGG-16	No Compression	32	534.74	1 ×
	Binary	2	41.67	13 ×
	Ternary	3	52.91	10.2 ×
	Logarithm	5	**73.13**	**7.3** ×
ResNet	No Compression	32	106.9	1 ×
	Binary	2	5.3	19.6 ×
	Ternary	3	7.4	14.4 ×
	Logarithm	5	**12.3**	**8.6** ×
DeepSpeech	No Compression	32	211	1 ×
	Binary	2	11.9	18.7 ×
	Ternary	3	14.4	14.5 ×
	Logarithm	5	**28.7**	**7.4** ×

momentum, Adam [70], and batch normalization [60].

4.4.2 Evaluation on the Impact of Gradient Quantization

The goal of the gradient quantization is to make the volume of message sent over network smaller in order to reduce the communication time in each iteration, and therefore to increase the training throughput during the whole training procedure.

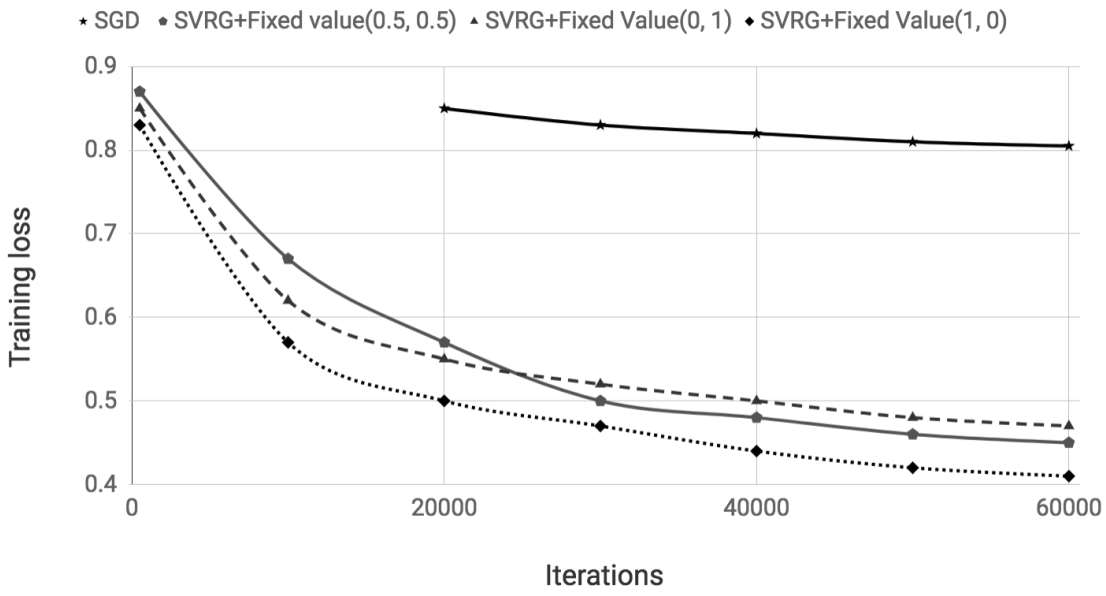
Table 4.2 presents the volume of gradient sent over network for different models. Communication ratio for various compression methods is reported. For logarithm quantization, we choose bit-width at 5, which is sufficient to produce a high prediction accuracy. The reduction ratio of logarithm compression is slightly worse than Binary and Ternary because the bit width we used is higher than the other two. Following experiment in next section will show that the sacrifice of the reduction ratio is compensated by a great improvement in prediction accuracy compared to the other two compression methods. It achieves nearly no loss in accuracy compared to traditional SGD training without compression at all. For logarithm compression, the reduction ratio is constantly around 8 × for all models. Reduction at this amount is good enough to relieve the

Table 4.3: Comparing training throughput on GPU cluster

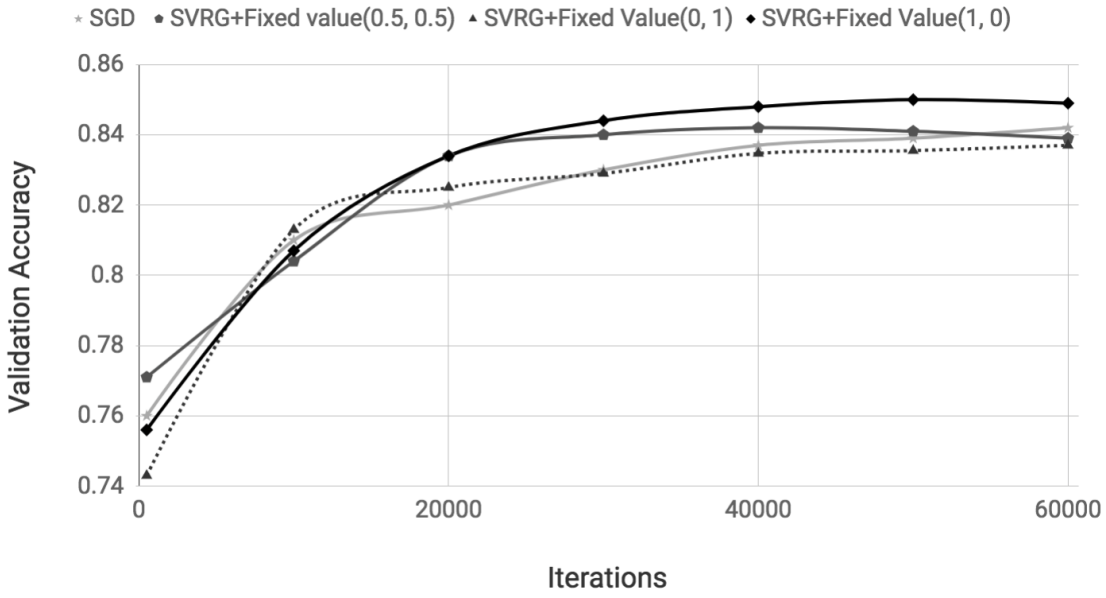
Network	Algorithm	Bit-width	Troughput (image/sec)	Improvement Ratio
AlexNet	No Compression	32	932	1 ×
	Binary	2	4247	4.6 ×
	Ternary	3	3560	3.8 ×
	Logarithm	5	**3258**	**3.5** ×
VGG-16	No Compression	32	74	1 ×
	Binary	2	773	10.5 ×
	Ternary	3	691	9.3 ×
	Logarithm	5	**602**	**8.2** ×
ResNet	No Compression	32	1285	1 ×
	Binary	2	3471	2.70 ×
	Ternary	3	2976	2.32 ×
	Logarithm	5	**2860**	**2.2** ×

problem of network congestion. Solving the congestion will lead to significant increase in system scalability and training throughput.

Model compression by using quantization techniques can reduce the communication overhead. Consequently, a high throughput improvement on the distributed GPU cluster is expected when training a deep neural network. Table 4.3 shows the result of training throughput improvement. We run this experiment with a cluster of 8 GPUs spread over three workers. The dataset used is Cifar-10. The improvement ratio for VGG-16 is obviously higher than that on AlexNet and ResNet. This is partially because the size of VGG-16 is larger than the other two. The larger the size, the easier the congestion happens when transferring gradients over network. This indicates the large the training model, the more benefits the training throughput would get by adopting quantization network. Note it is known the scalability of a distributed network is not linear to the throughput because the network overhead. This is determined by the communication-to-computation ratio of the deep neural network. Once the number of workers reaches a certain scalability bound, the congestion happens, then the performance drops significantly. However, resolving the congestion issue can improve the scalability to a much higher level. Moreover, in our experiment, the network bandwidth is around 6 Gbps. It can be expected the quantization technique will be more efficient when the network bandwidth becomes narrower.



(a) Training loss



(b) Validation Accuracy

Figure 4.13: Learning curves of SVRG and Fixed Coefficient Values on ResNet with Cifar10

Table 4.4: Comparison of gradient aggregation on ImageNet with AlexNet and VGG-16

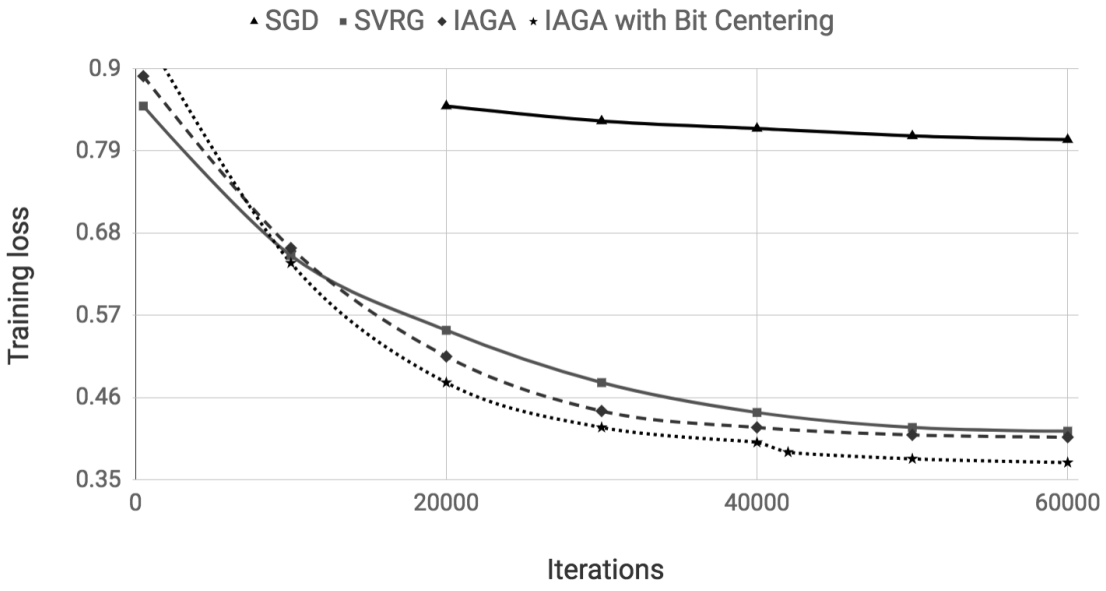
Network	Training Method	Top-1 accuracy	Top-5 accuracy	Batch Size
AlexNet	SGD	58.89%	80.23%	128
	SVRG+FV(0.5,0.5)	60.04%	81.44%	128
	SVRG+FV(0,1)	59.11%	80.67%	128
	SVRG+FV(1,0)	**60.26%**	**82.08%**	128
VGG-16	SGD	68.57%	88.87%	512
	SVRG+FV(0.5,0.5)	69.13%	89.21%	512
	SVRG+FV(0,1)	70.01%	90.19%	512
	SVRG+FV(1,0)	**71.12%**	**90.92%**	512

4.4.3 Evaluation on Gradient Aggregation with Various Precisions

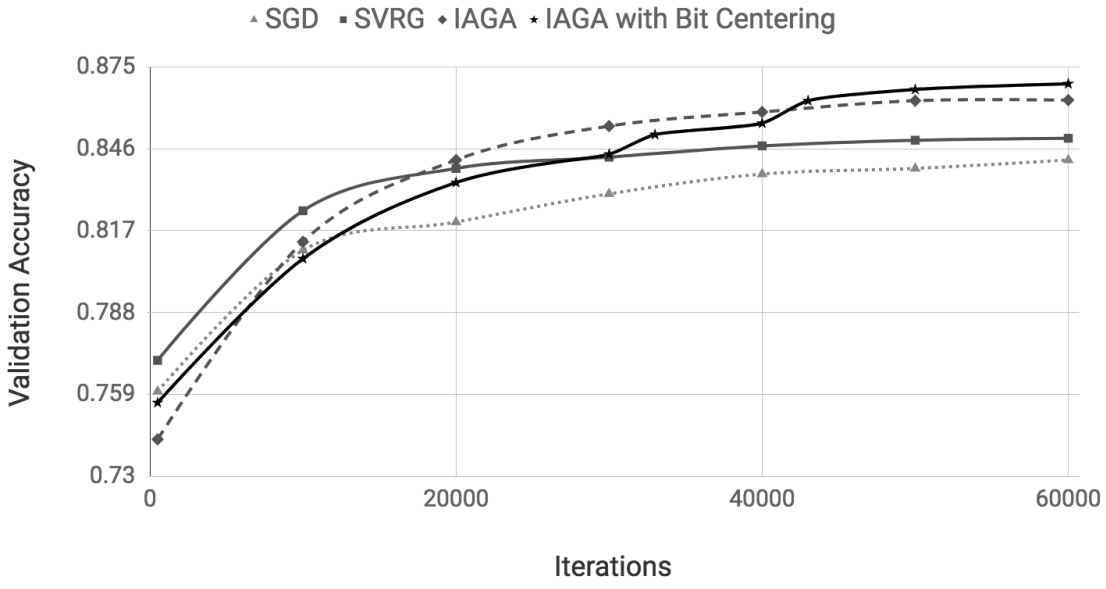
We evaluate the aggregation strategies in this section. In the aggregation step, the first operation is to calculate G_p and G_c. The second step is to choose coefficient for both of them for aggregation. We use vanilla SGD as the baseline and we report the performance of coefficient choosing strategies and our proposed algorithm IAGA.

In Figure 4.13, we plot the learning curve of SVRG with fixed value in choosing coefficient. The experiment was run with ResNet using Cifar10. Among three choices of fixed coefficient values, $\alpha = 1, \beta = 0$ performs the best in terms of the training speed and loss. It reaches the training loss of around 0.5 at iteration 20k, which is the earliest among three. And its final loss is 0.41, which is the lowest. $\alpha = 0.5, \beta = 0.5$ comes next to it. Finally, it is the $\alpha = 0, \beta = 1$. This can be explained using the description in Section 4.3.2. Recall that $\alpha = 0, \beta = 1$ is just a quantized training without any aggregation with higher precision values. The loss is therefore reduced by around 0.12. In the Figure 4.13b, by looking at the prediction accuracy on validation dataset, we can observe a similar pattern. Moreover, the increasing speed of $\alpha = 1, \beta = 0$ near the end of the training (iteration 40k to 50k) is rapid compared to other configurations. This demonstrates that training procedure can get real benefits from aggregating with high-precision gradients trained on masters. This acts as a fine tune and it is very important especially when the convergence speed slow down. We report the results of the same experiment setting but using different neural networks and datasets in Table 4.4.

We use 4 GPUs to run AlexNet with batch size of 128 and run VGG with 16 GPUs with batch size of 512. It shows similar result as Figure 4.13. It shows the combination of SVRG and linear



(a) Training loss



(b) Validation Accuracy

Figure 4.14: Learning curves of IAGA with and without bit centering on ResNet with Cifar10

Table 4.5: Evaluation of IAGA on ImageNet with AlexNet and VGG-16

Network	Training Method	Top-1 accuracy	Top-5 accuracy	Batch Size
AlexNet	SGD	58.89%	80.23%	128
	SVRG	60.44%	81.87%	128
	IAGA	**62.79%**	**83.16%**	128
	IAGA with BC	**63.03%**	**83.82%**	128
VGG-16	SGD	68.57%	88.87%	512
	SVRG	69.98%	89.58%	512
	IAGA	**70.34%**	**90.63%**	512
	IAGA with BC	**71.20%**	**91.17%**	512

combination in cooperated training can outperforms the vanilla SGD by at least 2% accuracy gain.

We evaluate the performance of our proposed algorithm IAGA in Figure 4.14. We compare it with SVRG and vanilla SGD. We also experiment IAGA with and without bit centering. Figure 4.14a shows that IAGA comes close to the training loss of full-precision SVRG, while it significantly outperforms vanilla SGD. IAGA with bit centering shows a slower decreasing rate in training loss at the beginning 10k iterations. As the training going on, bit centering technique gradually shrink the domain of gradient values, making the noise of gradients getting smaller in different training stages. Its training loss shows a small step down at iteration around 40k, indicating the bit centering is acting an important role in optimizing the gradient searching area. It achieves the smallest training loss compared to the other three. In terms of validation accuracy, Figure 4.14b shows that IAGA with bit centering produces a model with improved validation accuracy of 0.6% when compared to IAGA. Furthermore, IAGA with bit centering achieves competitive result when compared to SVRG. We report the results of the same experiment setting but using different neural networks and datasets in Table 4.5.

For speech recognition, we run DeepSpeech recurrent network on Librispeech ASR corpus dataset with 4 nodes. We report the word error rate (WER) in Table 4.6. The *WER clear* is clear speech without background noise while the *WER noisy* is noisy speech where the speaker's voice is blurred. The table shows the same improvement acquired from the proposed algorithm IAGA as for the convolutional neural networks. The neural networks trained with IAGA achieves higher speech recognition correction rate on both clear and noisy speech material. This is trained with a quantized gradients aggregation with compression ratio of around 20×. This demonstrates the

Table 4.6: Training results of speech recognition with RNN

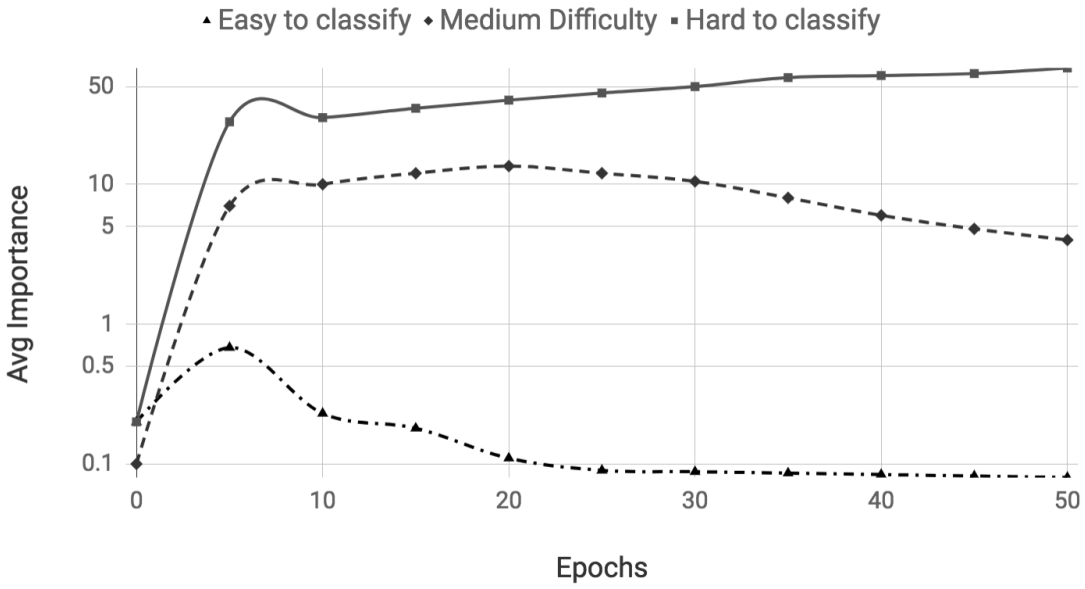
Network	Training Method	WER clear	WER noisy	Gradient Size (MB)
DeepSpeech 2	SGD	9.51%	27.24%	502
	SVRG	8.49%	27.03%	471
	IAGA	**8.03%**	**26.97%**	**23**
	IAGA with BC	**7.47%**	**25.1%**	**22.6**

capability of IAGA migrating from CNN to other popular RNNs, such as LSTM.

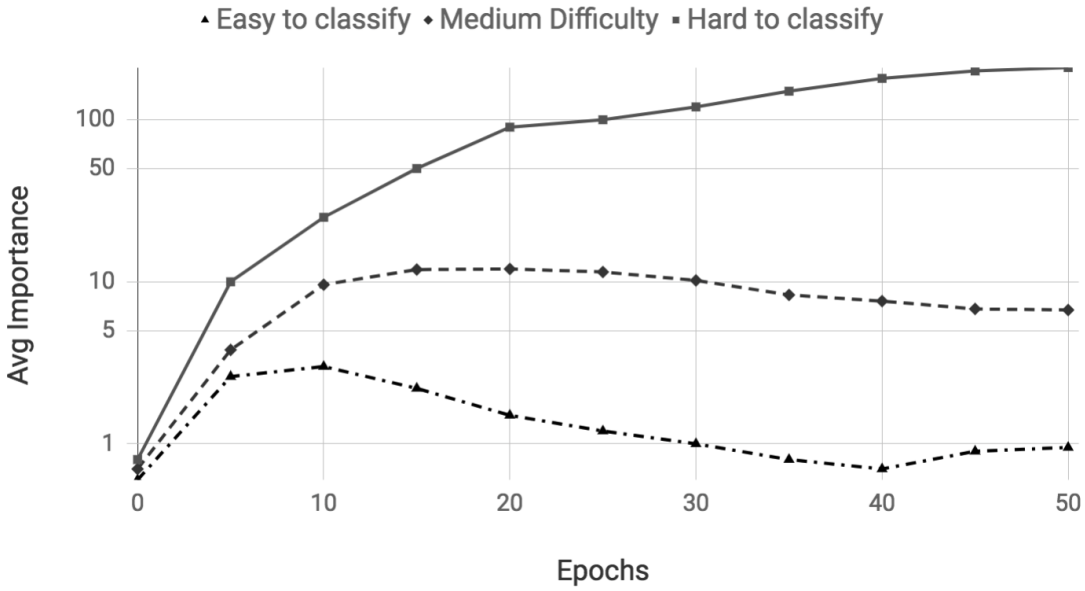
4.4.4 Evaluation of Importance Sampling on Training Batches

In this section, we evaluate the importance sampling to have a better understanding of how importance of different data points influence the training process. We use MNIST and Cifar-10 for empirical studies in this section due to their relatively small size compared to ImageNet. At first, we calculate the importance of each sample after each epoch. Next, we cluster all data points into several clusters based on their importance. We use negative log likelihood as the metric. Negative log likelihood has been used to measure how easy a sample is to be labeled with a class. Ideally, the larger the importance of a sample is, the larger its negative log likelihood is. But this is not always the case. So finally we choose to cluster based on importance but take negative log likelihood as a reference.

We use k-means to cluster the images into three clusters both on MNIST and Cifar-10. The three clusters are: 1) images that are easy to classify, therefore have small importance score; 2)images that are extremely hard to be classified, therefore have large importance score; 3) images that are not quite easy to classify, and their importance falls in between the first cluster and the second one. After labeling all images with one of the three classes, we keep monitoring the changing trend of their importance score and calculate an average on each cluster at the end of each epoch. We plot the trends in Figure 4.15. Because of the large span of average importance among three clusters, we use log scale to represent the vertical axis. We can see that on both Figure 4.15a and Figure 4.15b, the importance of images that are easy to classify increases rapidly at the initial several epochs then decreases at a similar rate after reaching the peak. This indicates that after the model parameters have been trained to classify these images with an acceptable prediction accuracy, these images do not contribute much the following training. On the contrary,



(a) MNIST



(b) Cifar-10

Figure 4.15: Trending of average importance of different types of images on MNIST and Cifar-10.

Table 4.7: Number of data points by hardness in classification on MNIST and Cifar-10

Dataset	# of easy to classify	# of medium difficulty	# of hard to classify	Ratio
MNIST	52175	6521	1304	40 : 5 : 1
Cifar-10	41668	7812	520	80 : 15 : 1

Table 4.8: Comparison of ISGD and SGD

Network	Datasets	Top-1 Acc.		Top-5 Acc.		Target Acc.	Iters to Target	
		ISGD	SGD	**ISGD**	SGD		**ISGD**	SGD
AlexNet	MNIST	**99.2%**	99.1%	**99.8%**	99.6%	99%	**20**	28
	CIFAR	**76.4%**	75.8%	**93.9%**	91.5%	75%	**39**	51
	ImageNet	**58.5%**	58.9%	**79.6%**	80.2%	78%	**74**	85
VGG-16	MNIST	**99.4%**	99.3%	**99.9%**	99.9%	99%	**7**	10
	CIFAR	**93.6%**	92.3%	**96.9%**	96.4%	90%	**25**	29
	ImageNet	**69.3%**	68.5%	**90.2%**	88.5%	65%	**61**	72
ResNet	MNIST	**99.4%**	99.3%	**99.9%**	99.9%	99%	**5**	8
	CIFAR	**94.6%**	94.2%	**99.1%**	99.3%	90%	**23**	31
	ImageNet	**76.7%**	75.3%	**94.1%**	93.3%	73%	**43**	69

the importance of the other two clusters of images keep increasing as the training going while the medium difficulty group slightly decreases when the training approaches the end. This shows that the model experience fine tuning on fitting images that are the hardest to classify. Intuitively, the gradients of these images need to be kept with a relatively higher precision in order to achieve a more accurate classification. This validates our design that sub-sampling a portion of more informative images to be trained on masters with higher precision.

We also find the number of images in each cluster is not equally distributed. That is the *easy to classify* class takes up the largest part of a dataset, followed by *medium difficulty* class. Images that are extremely hard to be classified only takes up a very tiny small part. We show these statistical information in Table 4.7.

Next, we train deep neural networks with importance sampling (ISGD) to evaluate its performance. We compare it with uniform sampling SGD (SGD) in terms of the time used to reach a certain accuracy on validation datasets. We run these experiment using ResNet on all three CNN datasets. We omit the training error plot and validation plot as they have similar trends with Figure 4.14. We detail the training results in Table 4.8. We can see that ISGD outperforms SGD

consistently in all experiment settings, which demonstrates the effectiveness of ISGD. In ImageNet experiment, ISGD has faster convergence rate than SGD by 14.6%. ISGD takes around 18 hours to reach the 73% top-1 accuracy while SGD takes more than 22 hours. In CIFAR experiment, ISGD has faster convergence rate than SGD by 23.7%. ISGD takes around 5 mins to reach the 75% top-1 accuracy while SGD takes 7 mins. Finally, in MNIST experiment, ISGD has faster convergence rate than SGD by 29.6%. ISGD takes around 35 seconds to reach the 99% top-1 accuracy while SGD takes around 1 min. Because the training has random initialization, we repeat the each experiment for 10 times and report the averages. The real performance may still be different. In summary, importance sampling can benefit the SGD training by significantly accelerating the whole procedure.

4.5 Summary

In this chapter, we designed a deep neural network (DNN) training paradigm that is not only suitable for hypergraph analysis, but a more general one that can be applied to many other application scenarios. Distributed hypergraph analysis using deep learning techniques will definitely benefit from this paradigm. We proposed a cooperated low precision training (C-LPT) paradigm for DNN training. In C-LPT, we allowed masters and workers to keep two different sets of a model in different precision level. In each training iteration, the workers are trained on a low-precision model using a large batch size, while masters are trained on a small portion of the batch (which are sampled from the large batch size trained on workers) with a high-precision model.

We investigated quantization methods and design a logarithmic quantization method with two factors. Instead of using full-precision (i.e., 32-bit floating points) representation, we restricted the values of parameters on workers to be either powers of two or zero. To minimize the error caused by quantization, a re-scaling strategy called *bit centering* [26] is integrated in our algorithm. In this way, the error of quantization will converge to zero asymptotically.

We explored approaches to reduce the variance in training data points and we extended C-LPT to adopt importance sampling for variance reduction. In C-LPT, importance sampling happens only when sampling a subset of the training batch on workers to be trained on the master side. This particular batch on the worker side is uniformly sampled from the whole dataset.

We conducted extensive experiments using C-LPT with various neural network architectures and real datasets. The results demonstrated that C-LPT can benefit DNN training in two folds. Firstly, the communication overhead is largely reduced as the bi-directional partial gradient updates between masters and workers are both in low-bits. Secondly, the noises introduced from the quantization and the variance of data points in a batch are both addressed elegantly as masters and workers are working in a cooperating manner to compensate for the loss of each other.

Chapter 5
Conclusions and Future Work

5.1 Conclusions

In this thesis, we focused on developing techniques for real-time hypergraph partitioning and quantized neural network training in the context of large scale and distributed data processing.

In Chapter 3, we proposed a real-time hypergraph partitioning algorithm, i.e., streaming refinement partitioning (SRP). We first explained why real-time partitioning is important and discussed its differences from the offline partitioning problem. In the streaming partitioning settings, vertices arrive one at a time in a sequential manner, and each one should be assigned to one partition in a very short time after it arrives. A near linear time algorithm is needed for the partitioning. Then, we identified the optimization targets when partitioning a hypergraph stream and formulated the problem. We provided theoretical analysis over the formulated problem and showed that hypergraph partitioning is an NP-hard problem. Streaming partitioning is even harder as less information is known when a new vertex is to be processed. After that, we proposed streaming refinement partitioning (SRP), which partitions a streaming hypergraph in two steps. In the first step, rough partitioning, we investigated a number of heuristics and chose to partition with a greedy strategy. In the second step, iterative refinement, we used label propagation with a fixed-size sliding window to constrain the partitioning time regardless of the streaming length. Finally, we implemented SRP on the HyperX framework and compared the performance of SRP with other online and offline partitioning algorithms. The results showed that by iteratively running refinement, the cut size becomes smaller than rough partitioning by at least 10% on both real datasets and synthetic datasets. SRP always outperforms offline partitioning algorithms in terms of workload balance and delivers up to 2.6 times speed-up compared with offline partitioning algorithms.

The results demonstrated that SRP not only provided better partitioned hypergraphs in terms of cut size and work-load balance, but also delivered more efficient and effective performance when running hypergraph learning algorithms.

In Chapter 4, we designed a cooperated low precision training (C-LPT) paradigm for deep neural network training. In C-LPT, we allowed masters and workers to keep two different sets of the same model in different precision levels. In each training iteration, the workers are trained with a low-precision model using a large batch size, while masters are trained on a small portion of the batch (which are sampled from the large batch used on workers) with a high-precision model. We first investigated a number of quantization methods and designed a logarithmic quantization method with two factors. Instead of using full-precision (i.e., 32-bit floating points) representation, we restricted the values of parameters on workers to be either powers of two or zero. Because of the characteristic of logarithm, the domain of representable values is not equally divided. Densities of values at two ends are much less than the density around zero. This uneven distribution, however, is a perfect match with the distribution of normalized gradients. To minimize the error caused by quantization, a re-scaling strategy called *bit centering* [26] was integrated in our algorithm. In this way, the error of quantization will converge to zero asymptotically. After that, we explored multiple approaches to reduce the variance in training data points and we extend C-LPT to adopt importance sampling for variance reduction. In C-LPT, importance sampling happens only when sampling a subset of the training batch on workers to be used on the master side. This particular batch on the worker side is still uniformly sampled from the whole dataset. Finally, we conducted extensive experiments using C-LPT with various neural network architectures and real datasets. The results demonstrated that C-LPT can benefit DNN training in two folds. Firstly, the communication overhead is largely reduced by around 10 times as the bi-directional partial gradient updates between masters and workers are both in low-bits. This results in a significant gain in training throughtput, where the improvement ratio is around 5. Secondly, the noises introduced from the quantization and the variance of data points in a batch are both addressed elegantly as masters and workers are working in a cooperating manner to compensate for the loss of each other. Therefore, the speed of convergence becomes faster. In experiments on ImageNet, Cifar, and MNIST, C-LPT with ISGD has faster convergence rate than SGD by 23.8% on average. In ImageNet experiment, C-LPT with ISGD takes around 18 hours to reach the 73% top-1 accuracy

while SGD takes more than 22 hours.

5.2 Future Work

There are several areas where we can improve our proposed algorithms.

- The streaming refinement partitioning algorithm treats each worker among the distributed cluster equally. This means that the algorithm does not distinguish the partitions on whether they locate on the same physical machine or not. The communication cost between workers on the same machine and different machines can be very different in real applications. To take this issue into consideration, the real-time partitioning algorithm should keep monitoring the network status between different partitions and design a finer cost function on optimizing the network communication, e.g., assigning different weights to different partitions as a regulation for the network.

- Following the above idea, to further optimize the allocation of replicas, we could take the network topological structure into consideration. When partitions are allocated to different workers but residing on the same physical machine, the data among these partitions can share the same memory. This could largely reduce the communication costs by avoiding unnecessary replicas. This is an ongoing research topic [124].

- The next step of quantized DNN training is towards unsupervised quantization or self-adaptive quantization. Our proposed C-LPT paradigm needs labeled data to retrain the network to determine the quantization hyper parameters and to keep a stable prediction accuracy. Quantization methods have a lot of hyper parameters to be determined before training starts, such as the sparsity of the network pruning and the bit-width. The selection of these hyper parameters is tedious but critical to the model performance. This normally requires considerable efforts as well as professional experience in tuning. Also, labeling large datasets is human resource intensive. Thus, to explore the methods that do not rely on human specified hyper parameters is a promising research topic. These problems may be tackled by unsupervised quantization or even fine-tuning-free quantization methods. One

possible direction may be to use reinforcement learning (RL) or generative adversarial networks (GAN).

- Distributed hypergraph processing is largely based on the Spark platform. On the other hand, distributed neural network training is based primarily on the Tensorflow framework. The connection between Spark and Tensorflow is mostly out of touch. The next research direction is to enable them to achieve seamless integration and retain their respective advantages. After they are integrated, data acquisition, processing, training, and classification can be done in a single pipeline.

Bibliography

[1] G. Alain, A. Lamb, C. Sankar, A. Courville, and Y. Bengio, "Variance reduction in sgd by distributed importance sampling," *arXiv preprint arXiv:1511.06481*, 2015.

[2] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen *et al.*, "Deep speech 2: End-to-end speech recognition in english and mandarin," in *Proceedings of International Conference on Machine Learning*, 2016, pp. 173–182.

[3] R. Andersen and K. J. Lang, "An algorithm for improving graph partitions," in *Proceedings of ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2008, pp. 651–660.

[4] K. Andreev and H. Racke, "Balanced graph partitioning," *Proceedings of Theory of Computing Systems*, vol. 39, no. 6, pp. 929–939, 2006.

[5] S. Anwar, K. Hwang, and W. Sung, "Fixed point optimization of deep convolutional neural networks for object recognition," in *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2015, pp. 1131–1135.

[6] M. Armbruster, "Branch-and-cut for a semidefinite relaxation of large-scale minimum bisection problems," 2007.

[7] M. Armbruster, M. Fügenschuh, C. Helmberg, and A. Martin, "A comparative study of linear and semidefinite branch-and-cut methods for solving the minimum graph bisection problem," in *Proceedings of International Conference on Integer Programming and Combinatorial Optimization*. Springer, 2008, pp. 112–124.

[8] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[9] S. T. Barnard and H. D. Simon, "Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems," *Proceedings of Concurrency: Practice and Experience*, vol. 6, no. 2, pp. 101–117, 1994.

[10] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *arXiv preprint arXiv:1308.3432*, 2013.

[11] P. Berenbrink, K. Khodamoradi, T. Sauerwald, and A. Stauffer, "Balls-into-bins with nearly optimal load distribution," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2013, pp. 326–335.

[12] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," *Proceedings of Society for Industrial and Applied Mathematics Review*, vol. 60, no. 2, pp. 223–311, 2018.

[13] F. Bourse, M. Lelarge, and M. Vojnovic, "Balanced graph edge partition," in *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2014, pp. 1456–1465.

[14] U. V. Catalyurek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *Proceedings of IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 7, pp. 673–693, 1999.

[15] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 2722–2730.

[16] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous sgd," *arXiv preprint arXiv:1604.00981*, 2016.

[17] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," in *Proceedings of European Conference on Computer Systems*. ACM, 2015, p. 1.

[18] R. Chen, J. Shi, B. Zang, and H. Guan, "Bipartite-oriented distributed graph partitioning for big learning," in *Proceedings of Asia-Pacific Workshop on Systems*. ACM, 2014, p. 14.

[19] W.-Y. Chen, Y. Song, H. Bai, C.-J. Lin, and E. Y. Chang, "Parallel spectral clustering in distributed systems," *Proceedings of the IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 3, pp. 568–586, 2011.

[20] W. Chen, J. Wilson, S. Tyree, K. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," in *Proceedings of International Conference on Machine Learning*, 2015, pp. 2285–2294.

[21] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 609–622.

[22] F. R. Chung, *Spectral graph theory*, 1997, no. 92.

[23] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *Proceedings of Journal of Machine Learning Research*, vol. 12, no. Aug, pp. 2493–2537, 2011.

[24] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Proceedings of Advances in Neural Information Processing Systems*, 2015, pp. 3123–3131.

[25] C. De Sa, M. Feldman, C. Ré, and K. Olukotun, "Understanding and optimizing asynchronous low-precision stochastic gradient descent," in *Proceedings of ACM Computer Architecture News*, vol. 45, no. 2. ACM, 2017, pp. 561–574.

[26] C. De Sa, M. Leszczynski, J. Zhang, A. Marzoev, C. R. Aberger, K. Olukotun, and C. Ré, "High-accuracy low-precision training," *arXiv preprint arXiv:1803.03383*, 2018.

[27] C. M. De Sa, C. Zhang, K. Olukotun, and C. Ré, "Taming the wild: A unified analysis of hogwild-style algorithms," in *Proceedings of Advances in Neural Information Processing Systems*, 2015, pp. 2674–2682.

[28] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *Proceedings of Advances in Neural Information Processing Systems*, 2012, pp. 1223–1231.

[29] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Proceedings of Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[30] A. Defazio, F. Bach, and S. Lacoste-Julien, "Saga: A fast incremental gradient method with support for non-strongly convex composite objectives," in *Proceedings of Advances in Neural Information Processing Systems*, 2014, pp. 1646–1654.

[31] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck, "Graph partitioning with natural cuts," in *Proceedings of IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2011, pp. 1135–1146.

[32] M. Deveci, K. Kaya, B. Uçar, and Ü. V. Çatalyürek, "Hypergraph partitioning for multiple communication cost metrics: Model and methods," *Proceedings of Journal of Parallel and Distributed Computing*, vol. 77, pp. 69–83, 2015.

[33] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek, "Parallel hypergraph partitioning for scientific computing," in *Proceedings of IEEE International Parallel & Distributed Processing Symposium*, 2006, pp. 10–pp.

[34] I. S. Dhillon, "Co-clustering documents and words using bipartite spectral graph partitioning," in *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2001, pp. 269–274.

[35] W. E. Donath and A. J. Hoffman, "Lower bounds for the partitioning of graphs," in *Proceedings of Selected Papers Of Alan J Hoffman: With Commentary*. World Scientific, 2003, pp. 437–442.

[36] Y. Dong, R. Ni, J. Li, Y. Chen, J. Zhu, and H. Su, "Learning accurate low-bit deep neural networks with stochastic quantization," *arXiv preprint arXiv:1708.01001*, 2017.

[37] G. Even, J. Naor, S. Rao, and B. Schieber, "Fast approximate graph partitioning algorithms," *Proceedings of SIAM Journal on Computing*, vol. 28, no. 6, pp. 2187–2214, 1999.

[38] Q. Fang, J. Sang, C. Xu, Y. Rui *et al.*, "Topic-sensitive influencer mining in interest-based social media networks via hypergraph learning." *Proceedings of IEEE Transactions of Multimedia*, vol. 16, no. 3, pp. 796–812, 2014.

[39] C. Farhat and M. Lesoinne, "Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics," *Proceedings of International Journal for Numerical Methods in Engineering*, vol. 36, no. 5, pp. 745–764, 1993.

[40] Y. Gao, M. Wang, D. Tao, R. Ji, and Q. Dai, "3-d object retrieval and recognition with hypergraph analysis," *Proceedings of IEEE Transactions on Image Processing*, vol. 21, no. 9, pp. 4290–4303, 2012.

[41] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," *arXiv preprint arXiv:1412.6115*, 2014.

[42] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: distributed graph-parallel computation on natural graphs." in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, vol. 12, no. 1, 2012, p. 2.

[43] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework." in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, vol. 14, 2014, pp. 599–613.

[44] W. W. Hager, D. T. Phan, and H. Zhang, "An exact algorithm for graph partitioning," *Proceedings of Mathematical Programming*, vol. 137, no. 1-2, pp. 531–556, 2013.

[45] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[46] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates *et al.*, "Deep speech: Scaling up end-to-end speech recognition," *arXiv preprint arXiv:1412.5567*, 2014.

[47] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[48] B. Hendrickson and R. Leland, "An improved spectral graph partitioning algorithm for mapping parallel computations," *Proceedings of SIAM Journal on Scientific Computing*, vol. 16, no. 2, pp. 452–469, 1995.

[49] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *Proceedings of IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[50] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," in *Proceedings of Advances in Neural Information Processing Systems*, 2013, pp. 1223–1231.

[51] L. Hou, Q. Yao, and J. T. Kwok, "Loss-aware binarization of deep networks," *arXiv preprint arXiv:1611.01600*, 2016.

[52] T. Hu, H. Xiong, W. Zhou, S. Y. Sung, and H. Luo, "Hypergraph partitioning for document clustering: A unified clique perspective," in *Proceedings of Special Interest Group on Information Retrieval*, 2008, pp. 871–872.

[53] J. Huang, "Similarity analysis with advanced relationships on big data," Ph.D. dissertation, 2015.

[54] Y. Huang, Q. Liu, and D. Metaxas, "] video object segmentation by hypergraph cut," in *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2009, pp. 1738–1745.

[55] Y. Huang, Q. Liu, S. Zhang, and D. N. Metaxas, "Image retrieval via probabilistic hypergraph ranking," in *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2010, pp. 3376–3383.

[56] Y. Huang and H. Lu, "Deep learning driven hypergraph representation for image-based emotion recognition," in *Proceedings of the ACM International Conference on Multimodal Interaction*. ACM, 2016, pp. 243–247.

[57] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Proceedings of Advances in Neural Information Processing Systems*, 2016, pp. 4107–4115.

[58] K. Hwang and W. Sung, "Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1," in *Proceedings of IEEE Workshop on Signal Processing Systems*. IEEE, 2014, pp. 1–6.

[59] T. Hwang, Z. Tian, R. Kuangy, and J.-P. Kocher, "Learning on weighted hypergraphs to integrate protein interactions and gene expressions for cancer outcome prediction," in *Proceedings of IEEE International Conference on Data Mining*, 2008, pp. 293–302.

[60] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[61] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the ACM International Conference on Multimedia*. ACM, 2014, pp. 675–678.

[62] R. Johnson and T. Zhang, "Accelerating stochastic gradient descent using predictive variance reduction," in *Proceedings of Advances in Neural Information Processing Systems*, 2013, pp. 315–323.

[63] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of ACM/IEEE International Symposium on Computer Architecture*. IEEE, 2017, pp. 1–12.

[64] G. Karypis and V. Kumar, "Metis–unstructured graph partitioning and sparse matrix ordering system, version 2.0," 1995.

[65] ——, "A fast and high quality multilevel scheme for partitioning irregular graphs," *Proceedings of SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[66] ——, "Multilevel k-way hypergraph partitioning," *Proceedings of Very Large Scale Integration Design*, vol. 11, no. 3, pp. 285–300, 2000.

[67] A. Katharopoulos and F. Fleuret, "Not all samples are created equal: Deep learning with importance sampling," *arXiv preprint arXiv:1803.00942*, 2018.

[68] J. Keuper and F.-J. Preundt, "Distributed training of deep neural networks: Theoretical and practical limits of parallel scalability," in *Proceedings of the Workshop on Machine Learning in High Performance Computing Environments*. IEEE Press, 2016, pp. 19–26.

[69] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: a system for dynamic load balancing in large-scale graph processing," in *Proceedings of ACM European Conference on Computer Systems*. ACM, 2013, pp. 169–182.

[70] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[71] S. Kirmani and P. Raghavan, "Scalable parallel graph partitioning," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 51.

[72] R. Krauthgamer, J. Naor, and R. Schwartz, "Partitioning graphs into balanced components," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2009, pp. 942–949.

[73] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.

[74] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of Advances in Neural Information Processing Systems*, 2012, pp. 1097–1105.

[75] C. Lanczos, *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*. United States Governm. Press Office Los Angeles, CA, 1950.

[76] A. H. Land and A. G. Doig, "An automatic method of solving discrete programming problems," *Proceedings of Econometrica: Journal of the Econometric Society*, pp. 497–520, 1960.

[77] Q. V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Y. Ng, "On optimization methods for deep learning," in *Proceedings of International Conference on Machine Learning*. Omnipress, 2011, pp. 265–272.

[78] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[79] S. Levine, P. Pastor, A. Krizhevsky, J. Ibarz, and D. Quillen, "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection," *Proceedings of the International Journal of Robotics Research*, vol. 37, no. 4-5, pp. 421–436, 2018.

[80] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server." in *Proceedings of Symposium on Operating Systems Design and Implementation*, vol. 14, 2014, pp. 583–598.

[81] M. Li, D. G. Andersen, A. J. Smola, and K. Yu, "Communication efficient distributed machine learning with the parameter server," in *Proceedings of Advances in Neural Information Processing Systems*, 2014, pp. 19–27.

[82] Q. Liao, J. Z. Leibo, and T. A. Poggio, "How important is weight symmetry in backpropagation?" in *Proceedings of Association for the Advancement of Artificial Intelligence*, 2016, pp. 1837–1844.

[83] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural networks with few multiplications," *arXiv preprint arXiv:1510.03009*, 2015.

[84] Q. Liu, Y. Huang, and D. N. Metaxas, "Hypergraph with sampling for image retrieval," *Proceedings of Pattern Recognition*, vol. 44, no. 10-11, pp. 2255–2262, 2011.

[85] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of ACM SIGMOD International Conference on Management of Data*. ACM, 2010, pp. 135–146.

[86] X. Martinez-Palau and D. Dominguez-Sal, "Analysis of partitioning strategies for graph processing in bulk synchronous parallel models," in *Proceedings of International Workshop on Cloud Data Management*. ACM, 2013, pp. 19–26.

[87] D. W. McDonald, "Recommending collaboration with social networks: a comparative evaluation," in *Proceedings of the Conference on Human Factors in Computing Systems*. ACM, 2003, pp. 593–600.

[88] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," *arXiv preprint arXiv:1603.01025*, 2016.

[89] J. Nishimura and J. Ugander, "Restreaming graph partitioning: simple versatile algorithms for advanced balancing," in *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2013, pp. 1106–1114.

[90] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur, "Librispeech: an asr corpus based on public domain audio books," in *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2015, pp. 5206–5210.

[91] J. Park, S. Li, W. Wen, P. T. P. Tang, H. Li, Y. Chen, and P. Dubey, "Faster cnns with direct sparse convolutions and guided pruning," *arXiv preprint arXiv:1608.01409*, 2016.

[92] P. Raghavendra, "Optimal algorithms and inapproximability results for every csp?" in *Proceedings of ACM Symposium on the Theory of Computing*, 2008, pp. 245–254.

[93] L. Ramaswamy, B. Gedik, and L. Liu, "A distributed approach to node clustering in decentralized peer-to-peer networks," *Proceedings of IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 9, pp. 814–829, 2005.

[94] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *Proceedings of European Conference on Computer Vision*. Springer, 2016, pp. 525–542.

[95] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Proceedings of Advances in Neural Information Processing Systems*, 2011, pp. 693–701.

[96] M. Schmidt, N. Le Roux, and F. Bach, "Minimizing finite sums with the stochastic average gradient," *Proceedings of Mathematical Programming*, vol. 162, no. 1-2, pp. 83–112, 2017.

[97] M. Sellmann, N. Sensen, and L. Timajev, "Multicommodity flow approximation used for exact graph partitioning," in *Proceedings of European Symposium on Algorithms*. Springer, 2003, pp. 752–764.

[98] N. Selvakkumaran and G. Karypis, "Multiobjective hypergraph-partitioning algorithms for cut and maximum subdomain-degree minimization," *Proceedings of IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, vol. 25, no. 3, pp. 504–517, 2006.

[99] R. O. Selvitopi, A. Turk, and C. Aykanat, "Replicated partitioning for undirected hypergraphs," *Proceedings of Journal of Parallel and Distributed Computing*, vol. 72, no. 4, pp. 547–563, 2012.

[100] Z. Shang and J. X. Yu, "Catch the wind: Graph workload balancing on cloud," in *Proceedings of IEEE International Conference on Data Engineering*. IEEE, 2013, pp. 553–564.

[101] J. Shi and J. Malik, "Normalized cuts and image segmentation," *Proceedings of IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 8, pp. 888–905, 2000.

[102] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *Proceedings of Nature*, vol. 529, no. 7587, p. 484, 2016.

[103] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *Proceedings of Nature*, vol. 550, no. 7676, p. 354, 2017.

[104] H. D. Simon, "Partitioning of unstructured problems for parallel processing," *Proceedings of Computing Systems in Engineering*, vol. 2, no. 2, pp. 135–148, 1991.

[105] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[106] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *Proceedings of the Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[107] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning." in *Proceedings of Association for the Advancement of Artificial Intelligence*, vol. 4, 2017, p. 12.

[108] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[109] H.-K. Tan, C.-W. Ngo, and X. Wu, "Modeling video hyperlinks with hypergraph for web video reranking," in *Proceedings of ACM Multimedia*, 2008, pp. 659–662.

[110] S. Tan, J. Bu, C. Chen, and X. He, "Using rich social media information for music recommendation via hypergraph model," in *Proceedings of Social Media Modeling and Computing*, 2011, pp. 213–237.

[111] S. Tan, Z. Guan, D. Cai, X. Qin, J. Bu, and C. Chen, "Mapping users across networks by manifold alignment on hypergraph." in *Proceedings of Association for the Advancement of Artificial Intelligence*, vol. 14, 2014, pp. 159–165.

[112] A. Trifunovic and W. J. Knottenbelt, "Parkway 2.0: A parallel multilevel hypergraph partitioning tool," *Proceedings of Lecture Notes in Computer Science*, pp. 789–800, 2004.

[113] A. Trifunović and W. J. Knottenbelt, "Parallel multilevel algorithms for hypergraph partitioning," *Proceedings of Journal of Parallel and Distributed Computing*, vol. 68, no. 5, pp. 563–581, 2008.

[114] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "Fennel: Streaming graph partitioning for massive scale graphs," in *Proceedings of ACM International Conference on Web search and data mining*. ACM, 2014, pp. 333–342.

[115] J. Ugander and L. Backstrom, "Balanced label propagation for partitioning massive graphs," in *Proceedings of ACM International Conference on Web Search and Data Mining*. ACM, 2013, pp. 507–516.

[116] L. G. Valiant, "A bridging model for parallel computation," *Proceedings of Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[117] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," in *Proceedings of Deep Learning and Unsupervised Feature Learning NIPS Workshop*, vol. 1. Citeseer, 2011, p. 4.

[118] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, and R. Fergus, "Regularization of neural networks using dropconnect," in *Proceedings of International Conference on Machine Learning*, 2013, pp. 1058–1066.

[119] F. Wang, J. Ye, W. Li, and G. Chen, "Is-asgd: Accelerating asynchronous sgd using importance sampling," *arXiv preprint arXiv:1706.08210*, 2017.

[120] Y. Wang, P. Li, and C. Yao, "Hypergraph canonical correlation analysis for multi-label classification," *Proceedings of Signal Processing*, vol. 105, pp. 258–267, 2014.

[121] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Proceedings of Advances in Neural Information Processing Systems*, 2016, pp. 2074–2082.

[122] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4820–4828.

[123] N. Xu, L. Chen, and B. Cui, "Loggp: a log-based dynamic graph partitioning method," *Proceedings of the Very Large Data Bases Endowment*, vol. 7, no. 14, pp. 1917–1928, 2014.

[124] J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai, "Seraph: an efficient, low-cost system for concurrent graph processing," in *Proceedings of International Symposium on High-performance Parallel and Distributed Computing*. ACM, 2014, pp. 227–238.

[125] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Proceedings of Knowledge and Information Systems*, vol. 42, no. 1, pp. 181–213, 2015.

[126] S. Yang, X. Yan, B. Zong, and A. Khan, "Towards effective partition management for large graphs," in *Proceedings of ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 517–528.

[127] W. Yang, G. Wang, L. Ma, and S. Wu, "A distributed algorithm for balanced hypergraph partitioning," in *Proceedings of IEEE Asia-Pacific Services Computing Conference*, 2016, pp. 477–490.

[128] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[129] H. Zha, X. He, C. Ding, H. Simon, and M. Gu, "Bipartite graph partitioning and data clustering," in *Proceedings of Conference on Information and Knowledge Management*, 2001, pp. 25–32.

[130] J. Zhang and I. Mitliagkas, "Yellowfin and the art of momentum tuning," *arXiv preprint arXiv:1706.03471*, 2017.

[131] W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-aware async-sgd for distributed deep learning," *arXiv preprint arXiv:1511.05950*, 2015.

[132] W. Zhao, S. Tan, Z. Guan, B. Zhang, M. Gong, Z. Cao, and Q. Wang, "Learning to map social network users by unified manifold alignment on hypergraph," *Proceedings of IEEE Transactions on Neural Networks and Learning Systems*, no. 99, pp. 1–13, 2018.

[133] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless cnns with low-precision weights," *arXiv preprint arXiv:1702.03044*, 2017.

[134] D. Zhou, J. Huang, and B. Schölkopf, "Learning with hypergraphs: Clustering, classification, and embedding," in *Proceedings of Advances in Neural Information Processing Systems*, 2007, pp. 1601–1608.

[135] J. Zhou and O. G. Troyanskaya, "Predicting effects of noncoding variants with deep learning–based sequence model," *Proceedings of Nature Methods*, vol. 12, no. 10, p. 931, 2015.

[136] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[137] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," *arXiv preprint arXiv:1612.01064*, 2016.

[138] L. Zhu, J. Shen, L. Xie, and Z. Cheng, "Unsupervised topic hypergraph hashing for efficient mobile image retrieval," *Proceedings of IEEE Transactions on Cybernetics*, vol. 47, no. 11, pp. 3941–3954, 2017.

[139] X. Zhu, Z. Ghahramani, and J. D. Lafferty, "Semi-supervised learning using gaussian fields and harmonic functions," in *Proceedings of International Conference on Machine Learning*, 2003, pp. 912–919.